Faster Circuits for And-Or Paths and Binary Addition

Dissertation

zur

Erlangung des Doktorgrades (Dr. rer. nat.)

der

Mathematisch-Naturwissenschaftlichen Fakultät

der

Rheinischen Friedrich-Wilhelms-Universität Bonn

Vorgelegt von

Anna Hermann

aus

Neuwied

Bonn, 05.05.2020

Angefertigt mit Genehmigung der Mathematisch-Naturwissenschaftlichen Fakultät der Rheinischen Friedrich-Wilhelms-Universität Bonn

Erstgutachter:	Herr Professor Dr. Stephan Held
Zweitgutachter:	Herr Professor Dr. Stefan Hougardy
Tag der Promotion:	10.07.2020
Erscheinungsjahr:	2020

Acknowledgments

At this point, I would like to take the opportunity to express my gratitude to many people:

First and foremost, I would like to thank my advisors Professor Dr. Stephan Held and Professor Dr. Stefan Hougardy for guiding and supporting me during the last years and for suggesting such a rich topic for my thesis. I am grateful to Professor Dr. Dr. h.c. Bernhard Korte and Professor Dr. Jens Vygen for providing excellent working conditions at the Research Institute for Discrete Mathematics.

I would like to thank the former BonnPlace team consisting of Dr. Ulrich Brenner, Nils Hoppmann, and Dr. Philipp Ochsendorf, who gave me a warm welcome at the institute and supported me during the last years. I am truly thankful for the day when Dr. Ulrich Brenner decided to join my work on logic optimization. Special thanks to him and Dr. Jannik Silvanus for the fruitful collaboration, which was both an inspiration and a pleasure. Thanks a lot to all my other colleagues for giving me an enjoyable time at the institute with many engaging discussions, in particular to Markus Ahrens, Dr. Pascal Van Cleeff, Siad Daboul, Benjamin Klotz, and Dr. Rudolf Scheifele.

Furthermore, I thank the past and present students of the BonnLogic team for their contributions to the project and for making my work more lively: Susanne Armbruster, Lucas Elbert, Falko Hegerfeld, Christian Nöbel, and Alexander Zorn.

Sincere thanks to Dr. Ulrich Brenner, Christian Nöbel, Dr. Matthäus Pawelczyk and Dr. Jannik Silvanus for proofreading parts of my thesis.

My deepest thanks go to my family – my parents Ellen and Albert Hermann, my sister Nicole Klier and my partner Jannik Silvanus – for always being there for me and for their unconditional support, in particular during the last months, when finishing this thesis was my only focus and I hardly took any time for my family.

I hope that we all get through the Corona crisis in good shape.

Contents

Chapter 1

Introduction

In this thesis, we consider the problem of computing fast circuits implementing certain Boolean functions. A **circuit** models the physical implementation of a Boolean function on a computer chip via elementary building blocks called **gates**. Mostly, given Boolean input variables $t_0, \dots, t_{m-1}$, we are interested in **And-Or path circuits**, i.e., circuits realizing functions of type

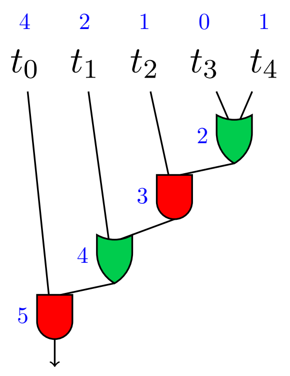
$$t_0 \lor \Biggl(t_1 \land \biggl(t_2 \lor \Bigl(t_3 \land \bigl(t_4 \lor (t_5 \land \dots)\bigr)\Bigr)\biggr)\Biggr).$$

Figure 1.1 shows two circuits on inputs $t_0, \dots, t_4$ which contain And (red) and Or (green) gates. The circuit C_1 is a classical And-Or path circuit – a path with gates alternating between And and Or. By comparing the Boolean functions $f(C_1)$ and $f(C_2)$ computed by C_1 and C_2, we see that C_1 and C_2 are logically equivalent:

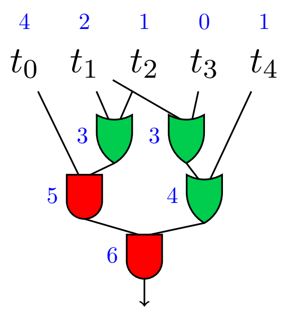
$$\begin{aligned} f(C_1) &= t_0 \land \Bigl(t_1 \lor \bigl(t_2 \land (t_3 \lor t_4)\bigr)\Bigr) \\ &= t_0 \land \Bigl((t_1 \lor t_2) \land \bigl(t_1 \lor (t_3 \lor t_4)\bigr)\Bigr) \\ &= \bigl(t_0 \land (t_1 \lor t_2)\bigr) \land \bigl((t_1 \lor t_3) \lor t_4\bigr) \\ &= f(C_2) \end{aligned}$$

Hence, C_2 is also an And-Or path circuit on the same inputs. Note that $\text{depth}(C_1) = 4$ and $\text{depth}(C_2) = 3$, where the **depth** of a circuit is the length of its longest path. Naturally, one is interested in fast circuits, and essentially, this means circuits with a low depth: Signals are propagated through the circuit, and a computation at a gate can only be performed once all signals are available.

In this model, it is implicitly assumed that all input signals are available at the same time. However, on a computer chip, this is rarely the case. Instead, usually, individual prescribed input **arrival times** are given. We recursively define the arrival time at a vertex as the maximum arrival time of its predecessors plus 1; and the **delay** of a circuit as maximum arrival time of any vertex. Given input variables $t_0, \dots, t_{m-1}$ and arrival times $a(t_0), \dots, a(t_{m-1}) \in \mathbb{N}$, the And-Or Path Circuit Optimization Problem asks for a delay-optimum And-Or path circuit on inputs $t_0, \dots, t_{m-1}$ with respect to the prescribed arrival times. Regarding the arrival times indicated in Figures 1.1(a) and 1.1(b), we have $\text{delay}(C_1) = 5$ and $\text{delay}(C_2) = 6$.



(a) We have $\operatorname{depth}(C_1) = 4$ and $\operatorname{delay}(C_1) = 5$.



(b) We have $\operatorname{depth}(C_2) = 3$ and $\operatorname{delay}(C_2) = 6$.

Figure 1.1: Two equivalent AND-OR path circuits C_1 and C_2.

Hence, depending on the arrival time profile, different circuits implementing a given Boolean function are preferable.

There are two main applications for the AND-OR PATH CIRCUIT OPTIMIZATION PROBLEM that are considered in this thesis: First, AND-OR path optimization can be used for logic restructuring of timing-critical paths in chip design. Secondly, the carry bits of adder circuits can naturally be computed via AND-OR paths.

Let us shortly explain the correlation of AND-OR paths and binary addition. Let two binary numbers $a = (a_0, \dots, a_{n-1})$ and $b = (b_0, \dots, b_{n-1})$ with bits ordered by increasing significance be given (note the difference to the AND-OR path index order). Then, in the standard method for binary addition, the **carry bits** are defined recursively by

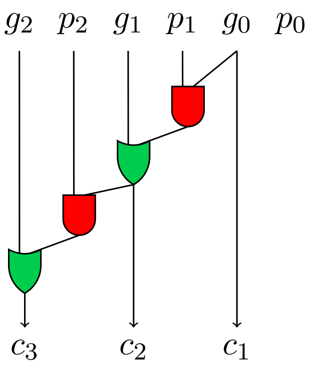
$$\begin{aligned} c_0 &= 0\,, \\ c_{i+1} &= (a_i \wedge b_i) \vee \Big((a_i \oplus b_i) \wedge c_i \Big) \quad \text{for } i \in \{0, \dots, n-1\}\,, \end{aligned}$$

where the term $a_i \oplus b_i$ is true if and only if exactly one of a_i and b_i is true. In other words, the carry bit c_{i+1} is true if and only if it is **generated** in position i as $g_i := a_i \wedge b_i$ is true, or it is carried over, also **propagated**, from position $i-1$ as c_i is true and $p_i := a_i \oplus b_i$ is true. As for $i \geq 0$, we have

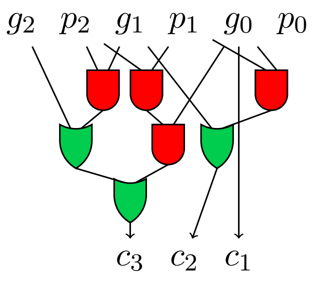
$$\begin{aligned} c_{i+1} &= (a_i \wedge b_i) \vee \Big((a_i \oplus b_i) \wedge c_i \Big) \\ &= \underbrace{(a_i \wedge b_i)}_{g_i} \vee \Bigg(\underbrace{(a_i \oplus b_i)}_{p_i} \wedge \bigg(\underbrace{(a_{i-1} \wedge b_{i-1})}_{g_{i-1}} \vee \Big(\underbrace{(a_{i-1} \oplus b_{i-1})}_{p_{i-1}} \wedge c_{i-1} \Big) \bigg) \Bigg) \\ &= g_i \vee \Bigg(p_i \wedge \bigg(g_{i-1} \vee \Big(p_{i-1} \wedge \dots \Big) \bigg) \Bigg) \end{aligned}$$

each carry bit c_{i+1} is actually an AND-OR path on inputs $g_i, p_i, \dots, p_1, g_0$. From the carry bits, the sum of a and b can be computed easily via

$$(a+b)_i = \begin{cases} c_i \oplus p_i & \text{if } i \in \{0, \dots, n-1\}\,, \\ c_n & \text{if } i = n\,. \end{cases}$$



(a) An adder circuit consisting of an And-Or path where all carry bits can be read off.



(b) An adder circuit composed of depth-optimum And-Or path circuits for each carry bit.

Figure 1.2: Two adder circuits for 3-bit binary numbers.

Thus, as in the literature, for us, an **adder circuit** is a circuit using only And2 and Or2 gates that computes all the carry bits based on the inputs $g_i, p_i, \ldots, p_1, g_0$. Figure 1.2 depicts two extreme types of adder circuits: The adder in Figure 1.2(a) has the minimum number of gates (also **size**) possible. The adder in Figure 1.2(b) has the minimum depth possible as it computes each carry bit by a depth-optimum And-Or path circuit. If circuit size is disregarded, the delay optimization problems for And-Or paths and adders are equivalent, so the main objective in adder optimization is computing all the carry bits with a reasonable total size.

In this thesis, we present algorithms for the computation of fast And-Or path and adder circuits regarding depth or delay optimization.

In Chapter 2, we introduce the mathematical objects studied in this thesis and give a survey on previous work about the optimization of And-Or path and adder circuits. In order to classify our results, we mention some of these statements here: A well-known lower bound on the depth of any binary circuit on $2n$ inputs is $\lceil \log_2(2n) \rceil$. Kraft's inequality [Kra49] yields a generalization of this lower bound with respect to input arrival times $a_0, \ldots, a_{n-1} \in \mathbb{N}$: A lower bound on the delay of any binary circuit is given by $\lceil \log_2 W \rceil$ where $W = \sum_{i=0}^{n} 2^{a_i}$. Moreover, Commentz-Walter [Com79] showed that there is an asymptotic lower bound of

$$\log_2 n + \log_2 \log_2 n + \text{const}$$

on the depth of And-Or path circuits and hence also for adder circuits.

In Chapter 3, we present a recursive algorithm for depth optimization of And-Or paths which computes linear-size And-Or path circuits with a depth of at most $\log_2 n + \log_2 \log_2 n + \text{const}$ in time $\mathcal{O}(n \log_2 n)$. These are the first known And-Or path circuits that, up to an additive constant, match the lower bound by Commentz-Walter [Com79] and, at the same time, have a linear size. The And-Or path circuits due to Grinchuk [Gri08] have a similar depth, but a size in the order of $\mathcal{O}(n \log_2 n)$.

In Chapter 4, we generalize our algorithm from Chapter 3 to delay optimization. Here, we construct And-Or path circuits with delay at most

$$\log_2 W + \log_2 \log_2 n + \log_2 \log_2 \log_2 n + \text{const}$$

and size at most $\mathcal{O}(n \log_2 n)$ in time $\mathcal{O}(n \log_2^2 n)$. This algorithm has already been published in Brenner and Hermann [BH19], but with a worse size and running time estimation. The previously best upper bound $\lceil \log_2 W \rceil + 2\sqrt{2 \log_2 n} + \text{const}$ on the delay was achieved by the circuits from Spirkl [Spi14].

In Chapter 5, we consider the delay optimization problem for generalized And-Or paths, a generalization of And-Or paths where gate types do not necessarily alternate. We present an exact algorithm with a running time of at most $\mathcal{O}(3^n)$ and, restricted to And-Or paths, of $\mathcal{O}\left(\left(\sqrt{6}\right)^n\right)$. For depth optimization of And-Or paths, we can further reduce the running time to $\mathcal{O}(n 2.02^n)$. Using sophisticated pruning techniques, we drastically improve our empirical running times. The only exact algorithms known so far consider the special case of depth optimization of And-Or paths, where the fastest algorithm is due to Hegerfeld [Heg18]. The largest instance we can solve has 64 inputs, while for Hegerfeld, it has 29 inputs. The running time of our algorithm is below 1.5 seconds for up to 60 inputs, and below 3 hours for the other instances. Based on our computations, we deduce the optimum depths of n-bit adder circuits for all $n \leq 8192$ that are a power of 2. To the best of our knowledge, for any $n \geq 32$, we are the first to discover and prove this result.

In Chapter 6, we present a dynamic program with running time $\mathcal{O}(n^4)$ for delay optimization of And-Or paths which has been published previously in Brenner and Hermann [BH20]. Our dynamic program fulfills the same delay guarantee as the theoretical algorithm from Chapter 4 and almost always computes delay-optimum solutions: Using our exact algorithm from Chapter 5, we demonstrate that on a testbed with 25000 And-Or path instances with up to 28 inputs, our dynamic program is delay-optimum on more than 95% of all instances, the maximum difference from the optimum delay is 1, and the average difference is 0.04. This is a significant improvement compared to the previously best implemented polynomial algorithms by Held and Spirkl [HS17b] and Rautenbach, Szegedy, and Werber [RSW06]: the circuit with best delay among their solutions is only optimum on 10% of the instances, deviates from the optimum by up to 4 and on average by 1.64.

Our dynamic program for delay optimization of And-Or paths is core routine of a logic restructuring framework called BonnLogic which has also been published in Brenner and Hermann [BH20]. BonnLogic is part of the BonnTools, a tool suite containing optimization algorithms for the design of computer chips. In IBM's industrial chip design flow, BonnLogic is applied to revise the logical structure of the most timing-critical paths. In Chapter 7, we describe BonnLogic in detail and demonstrate its efficiency and effectiveness on a testbed of recent 7nm chips.

Finally, in Chapter 8, we show an algorithm with running time $\mathcal{O}(n \log_2 n)$ for the construction of linear-size adder circuits. Its core routine is our depth optimization algorithm for And-Or paths from Chapter 3. Our linear-size adder circuits have a depth of at most

$$\log_2 n + \log_2 \log_2 n + \log_2 \log_2 \log_2 n + \text{const}\,,$$

which improves the best known upper bound on the depth of linear-size adder circuits. The previously best known linear-size adder circuits have a depth of at most $\log_2 n + 8\sqrt{\log_2 n} + 6 \log_2 \log_2 n + \text{const}$ and were published by Held and Spirkl [HS17a]. Hence, regarding the depth of linear-size adder circuits, we decreased the gap to the lower bound by Commentz-Walter [Com79] significantly from the order of $\mathcal{O}\left(\sqrt{\log_2 n}\right)$ to the order of $\mathcal{O}(\log_2 \log_2 \log_2 n)$.

Contributions

Several results of this thesis are joint work with Ulrich Brenner and Jannik Silvanus.

Regarding the depth optimization algorithm for And-Or paths in Chapter 3, the size optimization technique and analysis from Sections 3.3 and 3.4 has been developed jointly with Ulrich Brenner. He first proved that a circuit size of $\mathcal{O}(m \log_2 \log_2 m)$ can be obtained, and I refined his construction to yield a linear size of $27m$. In many iterations, we reduced the bound on the circuit size to $4.15m$.

Chapter 4 has been published previously in Brenner and Hermann [BH19], in concise form and with a worse analysis of circuit size and running time. The general idea for the delay optimization algorithm for And-Or paths and the proof of its delay guarantee via a statement like Theorem 4.1.2 are due to myself. Ulrich Brenner had the idea to strengthen the induction hypothesis as in Theorem 4.1.6 to circumvent the difficulties of the inductive proof of Theorem 4.1.2, see Remark 4.1.4. Together, we iteratively improved the result.

The exact algorithm for delay optimization of generalized And-Or paths in Chapter 5 is based on a new structural result, Theorem 5.2.9, which gives insights on the structure of certain delay-optimum solutions. The structure theorem has been discovered together with Ulrich Brenner and proven rigorously by myself. From this, the exact algorithm naturally follows. Running time analysis, practical implementation and speed-ups are joint work with Jannik Silvanus.

Chapters 6 and 7 have been developed by myself and have been published previously in concise form in Brenner and Hermann [BH20].

The adder optimization algorithm presented in Chapter 8 has been developed jointly with Ulrich Brenner. Together, we developed a first variant of the algorithm yielding linear-size adder circuits with a worse depth bound than presented in Theorem 8.3.6. Based on this, we alternatingly improved the result.

Chapter 2

Preliminaries

In this chapter, we present the main problems considered in this thesis. Sections 2.1 and 2.2 introduce the mathematical objects we will work on: Boolean functions, Boolean formulae, and circuits. In Section 2.3, we present several types of optimization problems related to these objects. In this thesis, we will be mostly interested in finding fast adder circuits and And-Or path circuits, and these special circuit classes are introduced in Sections 2.4 and 2.5. Section 2.6 surveys previous work regarding adder and And-Or path optimization.

2.1 Boolean Functions and Boolean Formulae

Our notation regarding Boolean functions is based on Crama and Hammer [CH11] and Savage [Sav98]. All results presented in this section can be found in Crama and Hammer [CH11] or Commentz-Walter [Com79], sometimes with different proofs.

We denote the set of natural numbers including zero by $\mathbb{N}$.

2.1.1 Basic Definitions

The most important basic objects considered in this thesis are Boolean functions. We will state introductory definitions before seeing some examples.

Definition 2.1.1. A **Boolean variable** is a variable assuming values in $\{0,1\}$. Given $n \in \mathbb{N}$, a **Boolean function** with n **Boolean input variables** (short, **inputs**) is a function $f\colon \{0,1\}^n \to \{0,1\}$. Every $\alpha \in \{0,1\}^n$ with $f(\alpha) = 1$ (respectively, $f(\alpha) = 0$) is a **true point** (respectively, **false point**) of f.

We will often view the values 1 and 0 as abstract symbols rather than integers. Then, we will also write **true** for 1 and **false** for 0, respectively.

Given a Boolean function $f\colon \{0,1\}^n \to \{0,1\}$, we will often denote the input variables by $x_0, \dots, x_{n-1}$ and abbreviatory write $x = (x_0, \dots, x_{n-1})$ for the ordered vector of inputs. With this, we can express the value of f on input variables $x = (x_0, \dots, x_{n-1})$ by $f\big((x_0, \dots, x_{n-1})\big) = f(x) \in \{0,1\}$.

Definition 2.1.2. Given a Boolean function $f\colon \{0,1\}^n \to \{0,1\}$, we call n the **arity** of f, and f an ***n*-ary** Boolean function. Given Boolean input variables $x \in \{0,1\}^n$, we also write $|x|$ for the number n of entries of x.

Definition 2.1.3. Let a Boolean function $f\colon \{0,1\}^n \to \{0,1\}$, an input x_i with $i \in \{0,\dots,n-1\}$, and a value $\alpha \in \{0,1\}$ be given. The **restriction** of f to $x_i = \alpha$ is the function $f\,|_{x_i=\alpha}\colon \{0,1\}^{n-1} \to \{0,1\}$ which is defined by

$$f\,|_{x_i=\alpha}\big((x_0,\dots,x_{i-1},x_{i+1},\dots,x_{n-1})\big) = f\big((x_0,\dots,x_{i-1},\alpha,x_{i+1},\dots,x_{n-1})\big)\,.$$

Definition 2.1.4. A Boolean function $f\colon \{0,1\}^n \to \{0,1\}$ **depends essentially** on an input x_i with $i \in \{0,\dots,n-1\}$ if $f\,|_{x_i=0}$ and $f\,|_{x_i=1}$ are different.

A common way to define a Boolean function is to provide its **truth table**, i.e., a list of all the points in $\{0,1\}^n$ together with their function values.

Definition 2.1.5. The Boolean function

$$f\colon \{0,1\}^2 \to \{0,1\},\ f\big((x_0,x_1)\big) = \begin{cases} 1 & \text{if } x_0 \neq x_1 \\ 0 & \text{otherwise} \end{cases}$$

is called **XOR function** or **Boolean exclusive disjunction**. We also write $f\big((x_0,x_1)\big) = x_0 \oplus x_1$.

A truth table for the XOR function is shown in Figure 2.1. Note that a Boolean function can also be defined by a complete list of its true points (or of its false points).

x_0	x_1	$f\big((x_0,x_1)\big)$
0	0	0
0	1	1
1	0	1
1	1	0

Figure 2.1: Truth table of XOR.

Another possibility to define a Boolean function is expressing it as a composition of elementary building blocks. We will do this in a recursive fashion based on the following three operations. Here, we consider Boolean variables as abstract symbols.

Definition 2.1.6. We define three operations on $\{0,1\}$.

The binary **AND** operation $\cdot \wedge \cdot\colon \{0,1\} \times \{0,1\} \to \{0,1\}$, also called **Boolean conjunction**, is defined by

$$x \wedge y = \begin{cases} 1 & \text{if } x = y = 1, \\ 0 & \text{otherwise.} \end{cases}$$

The binary **OR** operation $\cdot \vee \cdot\colon \{0,1\} \times \{0,1\} \to \{0,1\}$, also called **Boolean disjunction**, is defined by

$$x \vee y = \begin{cases} 1 & \text{if } x = 1 \text{ or } y = 1, \\ 0 & \text{otherwise.} \end{cases}$$

The unary **NOT** operation $\overline{\cdot}\colon \{0,1\} \to \{0,1\}$, also called **Boolean negation**, is defined by

$$\overline{x} = \begin{cases} 1 & \text{if } x = 0, \\ 0 & \text{otherwise.} \end{cases}$$

Note that if we interpret the elements of $\{0,1\}$ as numbers, the conjunction of $x, y \in \{0,1\}$ is actually the product of x and y, i.e., $x \wedge y = x \cdot y$.

We can view Boolean conjunction and disjunction as Boolean functions on 2 input variables and Boolean negation as a Boolean function on 1 input variable. The following definitions show how they can be used to describe more complex Boolean functions.

Definition 2.1.7. Given $n \in \mathbb{N}$ and Boolean variables $x_0, \dots, x_{n-1}$, a Boolean formula on the input variables $x_0, \dots, x_{n-1}$ is defined as follows:

(i) The constants $0, 1$ and the variables $x_0, \dots, x_{n-1}$ are Boolean formulae on $x_0, \dots, x_{n-1}$.

(ii) If ϕ and ψ are Boolean formulae on $x_0, \dots, x_{n-1}$, then $(\phi \vee \psi)$, $(\phi \wedge \psi)$ and $\overline{\phi}$ are Boolean formulae on $x_0, \dots, x_{n-1}$.

(iii) Any Boolean formula ϕ on $x_0, \dots, x_{n-1}$ arises from finitely many applications of the rules (i) and (ii).

We also write $\phi\big((x_0, \dots, x_{n-1})\big)$ or $\phi(x)$ to denote a Boolean formula on input variables $x = (x_0, \dots, x_{n-1})$, and call n the **arity of a** of ϕ.

We omit the parentheses if the formula is clear from the context.

Definition 2.1.8. Given Boolean input variables $x = (x_0, \dots, x_{n-1})$ and a Boolean formula $\phi(x)$, the Boolean function $f_\phi \colon \{0,1\}^n \to \{0,1\}$ **realized by** $\phi(x)$ is defined recursively as follows: For every point $(\alpha_0, \dots, \alpha_{n-1}) \in \{0,1\}^n$, the value of $f_\phi\big((\alpha_0, \dots, \alpha_{n-1})\big)$ is obtained by substituting α_i for x_i for all $i \in \{1, \dots, n\}$ in the formula ϕ and by recursively applying Definition 2.1.6 to compute the value of the resulting formula.

Given a Boolean function f realized by a Boolean formula ϕ, we call ϕ a **realization** of f.

In the following example, we will see realizations for the Xor function defined in Definition 2.1.5.

Example 2.1.9. Consider the Xor function $f \colon (x_0, x_1) \mapsto x_0 \oplus x_1$ from Definition 2.1.5. From the definition, we deduce the following two realizations of f:

$$\begin{aligned}\phi_1\big((x_0, x_1)\big) &= (x_0 \wedge \overline{x_1}) \vee (\overline{x_0} \wedge x_1)\\ \phi_2\big((x_0, x_1)\big) &= (x_0 \vee x_1) \wedge \overline{(x_0 \wedge x_1)}\end{aligned}$$

Here, we omit the redundant parentheses around the entire formula.

By Definition 2.1.8, a Boolean formula realizes a unique Boolean function, but as shown in Example 2.1.9, a Boolean function might have several realizations. For switching between different realizations, we list elementary properties of the operations defined in Definition 2.1.6.

Proposition 2.1.10. *Let Boolean variables $x, y, z \in \{0,1\}$ be given. Let $\circ$ denote any operation among $\wedge$ and $\vee$. The following properties hold:*

Commutativity:

$$x \circ y = y \circ x \tag{2.1}$$

Associativity:

$$(x \circ y) \circ z = x \circ (y \circ z) \tag{2.2}$$

Substitution of constants:

$$\begin{aligned} x \wedge 0 &= 0 & x \wedge 1 &= x \\ x \vee 0 &= x & x \vee 1 &= 1 \end{aligned} \tag{2.3}$$

Absorption rules:

$$\begin{aligned} x \wedge x &= x & x \wedge \overline{x} &= 0 \\ x \vee x &= x & x \vee \overline{x} &= 1 \\ x \vee (x \wedge y) &= x & x \wedge (x \vee y) &= x \end{aligned} \tag{2.4}$$

Distributivity:

$$x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z) \qquad x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z) \tag{2.5}$$

De Morgan's laws:

$$\overline{(x \wedge y)} = \overline{x} \vee \overline{y} \qquad \overline{(x \vee y)} = \overline{x} \wedge \overline{y} \tag{2.6}$$

□

Using these identities, we can now simplify Boolean functions that look complicated at first glance.

Example 2.1.11. For $\phi\big((x, y, z)\big) = \Big((x \wedge y) \vee \big((x \wedge \overline{y}) \wedge \overline{z}\big)\Big) \vee (x \wedge z)$, we have

$$\begin{aligned} \phi\big((x, y, z)\big) &= \Big((x \wedge y) \vee \big((x \wedge \overline{y}) \wedge \overline{z}\big)\Big) \vee (x \wedge z) \\ &\overset{(2.2)}{=} \Big((x \wedge y) \vee \big(x \wedge (\overline{y} \wedge \overline{z})\big)\Big) \vee (x \wedge z) \\ &\overset{(2.5)}{=} x \wedge \Big(\big(y \vee (\overline{y} \wedge \overline{z})\big) \vee z\Big) \\ &\overset{(2.5)}{=} x \wedge \Big(\big((y \vee \overline{y}) \wedge (y \vee \overline{z})\big) \vee z\Big) \\ &\overset{(2.4)}{=} x \wedge \big((y \vee \overline{z}) \vee z\big) \\ &\overset{\substack{(2.2)\\(2.4)}}{=} x\,. \end{aligned}$$

In this example, we have seen numerous Boolean formulae realizing the same Boolean function.

Definition 2.1.12. We call two Boolean formulae ϕ and ψ **equivalent** if ϕ and ψ realize the same Boolean function. If ϕ and ψ are equivalent, we write $\phi = \psi$.

In this thesis, we will often consider the problem of a finding Boolean formula with certain properties realizing a concrete Boolean function. A common way to solve this problem is to derive a Boolean formula from known Boolean formulae for similar functions. The following simple example illustrates this.

Example 2.1.13. By Example 2.1.9, we know that the Boolean formulae $\phi_1\big((x_0, x_1)\big) = (x_0 \wedge \overline{x_1}) \vee (\overline{x_0} \wedge x_1)$ and $\phi_2\big((x_0, x_1)\big) = (x_0 \vee x_1) \wedge \overline{(x_0 \wedge x_1)}$ both realize the Xor function $f\colon (x_0, x_1) \mapsto x_0 \oplus x_1$. Based on this, we want to find realizations for the Boolean function

$$g\big((x_0, x_1, x_2)\big) := \begin{cases} 1 & \text{if exactly one of } x_0, x_1 \text{ is true and } x_2 \text{ is true}\,, \\ 0 & \text{otherwise}\,. \end{cases}$$

The definition of g implies that for any realization ϕ of f, the Boolean formula

$$\psi\big((x_0, x_1, x_2)\big) := \phi\big((x_0, x_1)\big) \wedge x_2$$

is a realization of g. In particular, both

$$\psi_1\big((x_0, x_1, x_2)\big) := \phi_1\big((x_0, x_1)\big) \wedge x_2 = \big((x_0 \wedge \overline{x_1}) \vee (\overline{x_0} \wedge x_1)\big) \wedge x_2$$

and

$$\psi_2\big((x_0, x_1, x_2)\big) := \phi_2\big((x_0, x_1)\big) \wedge x_2 = \Big((x_0 \vee x_1) \wedge \overline{(x_0 \wedge x_1)}\Big) \wedge x_2$$

are realizations of g.

This concept is summarized more formally in the following remark.

Remark 2.1.14. Given Boolean functions $f, g, h\colon \{0,1\}^n \to \{0,1\}$ with $h(x) = f(x) \wedge g(x)$, any two realizations ϕ_f of f and ϕ_g of g yield a realization $\phi_h := \phi_f \wedge \phi_g$ of h. The same statement holds when replacing $\wedge$ by $\vee$; and given Boolean functions $f, g\colon \{0,1\}^n \to \{0,1\}$ with $f(x) = \overline{g}(x)$, any realization ϕ_g of g yields a realization $\phi_f := \overline{\phi_g}$ of f.

In order to avoid the notational overhead of switching between Boolean formulae and Boolean functions, the following remarks introduce simplified notation.

Remark 2.1.15. Consider two realizations ϕ, ψ of a Boolean function f. Since f is the unique Boolean function realized by ϕ and ψ by Definition 2.1.8 and we write $\phi = \psi$ for equivalent Boolean formulae, we may write $f = \phi$ and $f = \psi$.

Remark 2.1.16. Let $\phi\colon \{0,1\}^n \to \{0,1\}$ be a Boolean formula, and let $f_0, \ldots, f_{n-1}\colon \{0,1\}^k \to \{0,1\}$ be Boolean functions. If we identify ϕ with its realizing Boolean function f_ϕ as in Remark 2.1.15, we can define a Boolean function on k variables by

$$\alpha \mapsto f_\phi\big(f_0(\alpha), \ldots, f_{n-1}(\alpha)\big)$$

for each $\alpha \in \{0,1\}^k$, where the right-hand side simply uses function composition.

In particular, for Boolean functions $f, g\colon \{0,1\}^k \to \{0,1\}$, the functions $f \wedge g$, $f \vee g$, and $\overline{f}$ are defined by

$$\begin{aligned} (f \wedge g)(\alpha) &= f(\alpha) \wedge g(\alpha)\,, \\ (f \vee g)(\alpha) &= f(\alpha) \vee g(\alpha)\,, \\ \overline{f}(\alpha) &= \overline{f(\alpha)} \end{aligned}$$

for each $\alpha \in \{0,1\}^k$.

Remark 2.1.17. Note that Proposition 2.1.10 allows us to omit redundant brackets in the description of a Boolean function whenever we are not interested in the concrete representation by a Boolean formula. For example, we can extend the binary AND operation to an n-ary AND function by

$$\bigwedge_{i=0}^{n-1} x_i = x_0 \wedge x_1 \wedge \ldots \wedge x_{n-2} \wedge x_{n-1}\,, \tag{2.7}$$

where we may assume an arbitrary bracing due to associativity. Additionally, the commutativity rule (2.1) allows us to permute the variables arbitrarily in Equation (2.7). Analogously, we can define the n-ary OR function by

$$\bigvee_{i=0}^{n-1} x_i = x_0 \vee x_1 \vee \ldots \vee x_{n-2} \vee x_{n-1}\,, \tag{2.8}$$

Example 2.1.18. Remarks 2.1.14 and 2.1.15 allow us to formulate Example 2.1.13 in a much more compact way:

$$g\big((x_0, x_1, x_2)\big) = \Big(\big((x_0 \wedge \overline{x_1}) \vee (\overline{x_0} \wedge x_1)\big)\Big) \wedge x_2 = \Bigg(\Big((x_0 \vee x_1) \wedge \overline{(x_0 \wedge x_1)}\Big)\Bigg) \wedge x_2\,.$$

2.1.2 Normal Forms and Monotonicity

It is well-known that for every Boolean function f, there is a Boolean formula realizing f. We prove this statement via prime implicants.

Definition 2.1.19. Let $f\colon \{0,1\}^n \to \{0,1\}$ be a Boolean function with inputs $x = (x_0, \ldots, x_{n-1})$. A **literal** of f is a possibly negated input variable of f, i.e., x_i or $\overline{x_i}$ for some $i \in \{0, \ldots, n-1\}$. Consider a Boolean formula of the form $\iota\big((x_0, \ldots, x_{n-1})\big) = l_{i_1} \wedge \ldots \wedge l_{i_k}$, where $l_{i_1}, \ldots, l_{i_k}$ are literals of f. The formula ι is an **implicant** of f if for any $\alpha \in \{0,1\}^n$ with $\iota(\alpha) = 1$, we have $f(\alpha) = 1$. We write $\mathrm{lit}(\iota) = \big\{l_{i_1}, \ldots, l_{i_k}\big\}$ for the set of literals of ι. We call ι a **prime implicant** of f if there is no other implicant of f with $\mathrm{lit}(\pi) \subsetneq \mathrm{lit}(\iota)$. The set of all prime implicants of f is denoted by $\mathrm{PI}(f)$.

Observation 2.1.20. Let $f\colon \{0,1\}^n \to \{0,1\}$ be a Boolean function on inputs $x_0, \ldots, x_{n-1}$. Then, for any input x_i with $i \in \{0, \ldots, n-1\}$, the function f depends essentially on x_i if and only if there is a prime implicant of f containing x_i.

Theorem 2.1.21. *Let $f\colon \{0,1\}^n \to \{0,1\}$ be a Boolean function. Then, the Boolean formula $\phi(x) = \bigvee_{\pi \in \mathrm{PI}(f)} \pi(x)$ is a realization of f.*

Proof. Let $\alpha \in \{0,1\}^n$. We check that α is a true point of f if and only it is a true point of f_ϕ.

If $\phi(\alpha) = 1$, then there is some $\pi \in \mathrm{PI}(f)$ with $\pi(\alpha) = 1$. As π is a prime implicant of f, this implies that $f(\alpha) = 1$.

Now assume that $f(\alpha) = 1$. Consider the Boolean formula

$$\iota(x) = \bigwedge_{i:\alpha_i = 1} x_i \wedge \bigwedge_{i:\alpha_i = 0} \overline{x_i}\,.$$

Then, ι is an implicant of f. Let π be a prime implicant of f whose literals are a subset of the literals of ι. Then, $\pi(\alpha) = 1$, hence $\phi(\alpha) = 1$. $\square$

Hence, every Boolean function has a realizing Boolean formula. Furthermore, we can deduce the following statement.

Corollary 2.1.22. *A Boolean function is uniquely determined by the set of its prime implicants.* □

The realization of f given by Theorem 2.1.21 is an example for a disjunctive normal form of f. More generally, a **disjunctive normal form** (DNF) of f is a realization of the form

$$\phi\big((x_0,\dots,x_{n-1})\big) = \bigvee_{k=0}^{m-1}\left(\bigwedge_{i\in A_k} x_i \wedge \bigwedge_{i\in B_k}\overline{x_i}\right) \tag{2.9}$$

with $m \in \mathbb{N}$ and $A_k, B_k \subseteq \{0,\dots,n-1\}$ with $A_k \cap B_k = \emptyset$ for each $k = 0,\dots,m-1$. A **conjunctive normal form** (CNF) of f is a realization of the form

$$\phi\big((x_0,\dots,x_{n-1})\big) = \bigwedge_{k=0}^{m-1}\left(\bigvee_{i\in A_k} x_i \vee \bigvee_{i\in B_k}\overline{x_i}\right)$$

with $m \in \mathbb{N}$ and $A_k, B_k \subseteq \{0,\dots,n-1\}$ with $A_k \cap B_k = \emptyset$ for each $k = 0,\dots,m-1$. The following theorem states that every Boolean function has a conjunctive normal form. It is not hard to give a proof for this similar to the proof of Theorem 2.1.21, but at the end of Section 2.1.3, we will be able to provide a more elegant proof.

Theorem 2.1.23. *Any Boolean function $f\colon \{0,1\}^n \to \{0,1\}$ can be realized by a disjunctive normal form and by a conjunctive normal form.*

Given $\alpha, \beta \in \{0,1\}^n$, we write $\alpha \le \beta$ if $\alpha_i \le \beta_i$ for all $i \in \{0,\dots,n-1\}$. Using this notation, we can define monotone Boolean functions.

Definition 2.1.24. A Boolean function $f\colon \{0,1\}^n \to \{0,1\}$ is **monotone** if for all $\alpha, \beta \in \{0,1\}^n$ with $\alpha \le \beta$, we have $f(\alpha) \le f(\beta)$. A Boolean formula ϕ is **monotone** if it does not contain any negations.

Most functions considered in this thesis are monotone. In Corollary 2.1.26, we shall see that for any monotone Boolean function, there is a representing monotone Boolean formula, so the two apparently very different definitions of monotone functions and formulae fit together.

Lemma 2.1.25 (Crama and Hammer [CH11])**.** *Let $f\colon \{0,1\}^n \to \{0,1\}$ be a monotone Boolean function. Then, every prime implicant of f is monotone.*

Proof. Assume that there is a prime implicant π of f of the form $\pi = \overline{x_i} \wedge l_1 \wedge \ldots \wedge l_k$ for some $i \in \{0,\dots,n-1\}$ and $l_1,\dots,l_k$ being literals different from x_i and $\overline{x_i}$. By monotonicity of f, the formula $\pi' = x_i \wedge l_1 \wedge \ldots \wedge l_k$ is also an implicant of f. But then the formula

$$\pi'' := \pi \vee \pi' = (\overline{x_i} \wedge l_1 \wedge \ldots l_k) \vee (x_i \wedge l_1 \wedge \ldots \wedge l_k) = l_1 \wedge \ldots \wedge l_k$$

is another implicant of f, contradicting π being a prime implicant of f. □

Together with Theorem 2.1.21, this lemma implies the following statement.

Corollary 2.1.26 (Crama and Hammer [CH11]). *For any monotone Boolean function, there is a monotone Boolean formula.* □

However, a non-monotone Boolean formula may still realize a monotone Boolean function, see, e.g., Example 2.1.11.

For monotone Boolean functions, function decomposition and (prime) implicants behave in a canonical way.

Lemma 2.1.27. *Consider Boolean functions $f, g, h\colon \{0,1\}^n \to \{0,1\}$ with $f = g \vee h$. Then, any implicant of g or h is an implicant of f. Furthermore, if f, g, h are all monotone, then any (prime) implicant of f is an (prime) implicant of g or of h.*

Proof. Let $f, g, h\colon \{0,1\}^n \to \{0,1\}$ be Boolean functions on input variables $x_0, \ldots, x_{n-1}$, and let $\iota = l_{i_1} \wedge \ldots \wedge l_{i_k}$ be an implicant of g. For any $\alpha = (\alpha_0, \ldots, \alpha_{n-1}) \in \{0,1\}^n$ with $\alpha_{i_1} \wedge \ldots \wedge \alpha_{i_k} = 1$, we have $g(\alpha) = 1$, so $f(\alpha) = g(\alpha) \vee h(\alpha) = 1 \vee h(\alpha) = 1$. Thus, ι is an implicant of f. By symmetry of OR, the same holds for implicants of h.

Assume additionally that f, g, h are monotone and let κ be an implicant of f. Assume that κ is not an implicant of g or h. Then, there are $\alpha_g, \alpha_h \in \{0,1\}^n$ with

$$\kappa(\alpha_g) = \kappa(\alpha_h) = 1 \text{ and} \tag{2.10}$$

$$g(\alpha_g) = h(\alpha_h) = 0. \tag{2.11}$$

Define $\alpha \in \{0,1\}^n$ by $\alpha_i = \alpha_{g_i} \wedge \alpha_{hi}$ for $i \in \{0, \ldots, n-1\}$. Now, as κ is a product of literals, Equation (2.10) implies $\kappa(\alpha) = 1$. Furthermore, as g and h are monotone and $\alpha \leq \alpha_g, \alpha_h$, Equation (2.11) implies $g(\alpha) = f(\alpha) = 0$ and thus $f(\alpha) = g(\alpha) \vee h(\alpha) = 0$, which contradicts κ being an implicant of f.

Hence, κ is an implicant of g or h, without loss of generality of g. Now assume that κ is a prime implicant of f, but not a prime implicant of g. Then, there is an implicant λ of g whose literals are all contained in κ. But by the first statement of this lemma, λ is an implicant of f, contradicting to the assumption that κ is a prime implicant of f. Thus, κ is a prime implicant of g. □

For non-monotone functions, the second statement of this lemma does not hold: E.g., for the function $f_\phi\big((x,y,z)\big) = \Big((x \wedge y) \vee \big((x \wedge \overline{y}) \wedge \overline{z}\big)\Big) \vee (x \wedge z) = x$ from Example 2.1.11, x is an implicant of f_ϕ, but not of $(x \wedge y) \vee \big((x \wedge \overline{y}) \wedge \overline{z}\big)$ or $x \wedge z$.

2.1.3 Duality

Proposition 2.1.10 is invariant under the following operation: Exchange all $\wedge$ and $\vee$ gates, and exchange all 1 and 0 symbols. This remarkable concept is called **duality**.

Definition 2.1.28. Let ϕ be a Boolean formula. The **dual Boolean formula** ϕ^* of ϕ can be obtained from ϕ by interchanging all $\wedge$ and $\vee$ operations, and all 1 and 0 symbols.

Definition 2.1.29. Let $f\colon \{0,1\}^n \to \{0,1\}$ be a Boolean function. The **dual Boolean function** $f^*\colon \{0,1\}^n \to \{0,1\}$ of f is defined by

$$f^*\big((x_0, \ldots, x_{n-1})\big) = \overline{f\big((\overline{x_0}, \ldots, \overline{x_{n-1}})\big)}\,.$$

Abusing notation, we write $\overline{x} := (\overline{x_0}, \ldots, \overline{x_{n-1}})$ and thus $f^*(x) = \overline{f(\overline{x})}$.

Dualization of Boolean formulae and functions is an involution, as can directly be seen from Definitions 2.1.28 and 2.1.29.

Proposition 2.1.30. *Given a Boolean formula ϕ, we have $(\phi^*)^* = \phi$. Given a Boolean function f, we have $(f^*)^* = f$.* □

The following theorem shows why Definition 2.1.29 is the appropriate way to define a dual Boolean function.

Theorem 2.1.31. *Let ϕ by a Boolean formula. If ϕ is a realization of the Boolean function f_ϕ, then ϕ^* is a realization of f_ϕ^*.*

Proof. Let $r \in \mathbb{N}$ denote the total number of operations in ϕ. We prove the statement by induction on r.

If $r = 0$, then ϕ is either a constant or a variable, and the definitions yield $\phi^*(0) = 1 = \overline{0} = f^*(0)$, and $\phi^*(1) = 0 = \overline{1} = f^*(1)$, and $\phi^*(x) = x = \overline{\overline{x}} = \overline{f(\overline{x})}$.

Now assume that $r > 0$, i.e., we can write $\phi = \psi \wedge \rho$, $\phi = \psi \vee \rho$ or $\phi = \overline{\psi}$ for some Boolean formulae ψ and ρ, and we can inductively assume that the statement holds for ψ and ρ.

If $\phi = \overline{\psi}$, we have $\phi^* = \left(\overline{\psi}\right)^* = \overline{\psi^*}$ by Definition 2.1.28, and ϕ^* realizes f_ϕ^* since

$$\begin{aligned}
f_{\overline{\psi^*}}(x) &\overset{\text{Rem. 2.1.14}}{=} \overline{f_{\psi^*}(x)} \\
&\overset{\substack{\text{Prop. 2.1.30,}\\ \text{(IH)}}}{=} \overline{f_\psi^*(x)} \\
&\overset{\text{Def. 2.1.29}}{=} f_\psi(\overline{x}) \\
&\overset{\text{Rem. 2.1.14}}{=} \overline{f_\phi(\overline{x})} \\
&\overset{\text{Def. 2.1.29}}{=} f_\phi^*(x)\,.
\end{aligned}$$

Otherwise, if

$$\phi = \psi \wedge \rho\,, \tag{2.12}$$

we have

$$f_\phi = f_\psi \wedge f_\rho \tag{2.13}$$

by Remark 2.1.14. Hence, we have

$$\phi^*(x) \overset{(2.12)}{=} (\psi \wedge \rho)^*(x) \overset{\text{Def. 2.1.28}}{=} \big(\psi^* \vee \rho^*\big)(x)\,.$$

By induction hypothesis and Remark 2.1.14, $\phi^* = \psi^* \vee \rho^*$ realizes the function $f_\psi^* \vee f_\rho^*$. But

$$\begin{aligned}
\left(f_\psi^* \vee f_\rho^*\right)(x) &\overset{\text{Rem. 2.1.16}}{=} f_\psi^*(x) \vee f_\rho^*(x) \\
&\overset{\text{Def. 2.1.29}}{=} \overline{f_\psi(\overline{x})} \vee \overline{f_\rho(\overline{x})} \\
&\overset{(2.6)}{=} \overline{f_\psi(\overline{x}) \wedge f_\rho(\overline{x})} \\
&\overset{(2.13)}{=} \overline{f_\phi(\overline{x})} \\
&\overset{\text{Def. 2.1.29}}{=} f_\phi^*(x)\,.
\end{aligned}$$

Note that the crucial step here was applying De Morgan's laws (2.6). Hence, ϕ^* realizes f_ϕ^*.

If $\phi = \psi \vee \rho$, the proof works analogously by exchanging all $\wedge$ and $\vee$ operations.

This proves the induction step and hence the theorem. □

In particular, we can deduce the following statement about dual formulae.

Corollary 2.1.32. *If ϕ and ψ are equivalent Boolean formulae, then the dual formulae ϕ^* and ψ^* are also equivalent.*

Proof. Since ϕ and ψ are equivalent, they realize the same Boolean function f. By Theorem 2.1.31, both ϕ^* and ψ^* realize f^* and hence are equivalent. $\square$

Using the concept of duality, we can now prove Theorem 2.1.23.

Proof of Theorem 2.1.23. Given a Boolean function $f\colon \{0,1\}^n \to \{0,1\}$, a disjunctive normal form of f exists by Theorem 2.1.21. In order to show that there is a conjunctive normal form of f, consider the dual function f^* of f. Again, Theorem 2.1.21 yields a disjunctive normal form ϕ for f^*. By Theorem 2.1.31, the dual formula ϕ^* realizes $(f^*)^* \overset{\text{Prop. 2.1.30}}{=} f$. This is a conjunctive normal form of f. $\square$

2.2 Circuits

A **circuit** is a model for the physical implementation of a Boolean function on a computer chip. Usually, a small set of building components called **gates** implementing elementary Boolean functions is available on a chip. By combining these building components to **circuits**, more complicated Boolean functions can be implemented. Note that this is very similar to the concept of Boolean formulae (cf. Definition 2.1.7) that are decomposed of the operations defined in Definition 2.1.6. Our notation related to graph theory is based on Korte and Vygen [KV18].

Definition 2.2.1. A **basis** is a set Ω of Boolean formulae. Each element $\phi \in \Omega$ is called a **gate**.

Definition 2.2.2. A **circuit** $C = (\mathcal{V}, \mathcal{E})$ over the basis Ω is an acyclic directed graph with labeled vertices $\mathcal{V} = \mathcal{I} \,\dot\cup\, \mathcal{G}$ such that the following conditions are satisfied:

- Each vertex $v \in \mathcal{I}$ fulfills $\delta^-(v) = \emptyset$ and $\delta^+(v) \neq \emptyset$ and is labeled either with a distinct Boolean variable x_v or with a constant (0 or 1). The vertices in $\mathcal{I}$ are called **inputs** of C.
- Each vertex $v \in \mathcal{G}$ fulfills $k := |\delta^-(v)| \geq 1$ and is labeled with a k-ary gate $\phi \in \Omega$ together with a fixed ordering $v_0, \dots, v_{k-1}$ of the predecessors $\delta^-(v)$ of v. The vertices in $\mathcal{G}$ are called **gates** of C. We denote the **gate type** $\phi \in \Omega$ of a gate vertex $v \in \mathcal{G}$ by $\operatorname{gt}(v) := \phi \in \Omega$.
- There is a subset $\emptyset \subsetneq \mathcal{O} \subseteq \mathcal{V}$ that we call the set of **outputs** of C. We demand each vertex with $\delta^+(v) = \emptyset$ to be an output, but there may be other outputs.

Given a circuit C, we also write $\mathcal{V}(C)$, $\mathcal{E}(C)$, $\mathcal{I}(C)$, $\mathcal{O}(C)$, $\mathcal{G}(C)$ for its nodes, edges, inputs, outputs, and gates, respectively.

Given a circuit C over a basis Ω and a vertex $v \in \mathcal{V}(C)$, the **Boolean formula $\boldsymbol{\phi_v}$ corresponding to $\boldsymbol{v}$** is defined recursively as follows:

- If $v \in \mathcal{I}(C)$, then $\phi_v = x_v$.
- Consider $v \in \mathcal{G}(C)$. Let $\phi \in \Omega$ denote the gate associated with v, and $v_0, \dots, v_{k-1}$ the ordered predecessors of v. Then $\phi_v = \phi(\phi_{v_0}, \dots, \phi_{v_{k-1}})$.

Now assume that $|\mathcal{O}(C)| = 1$. Then, we denote the unique output of C by $\mathrm{out}(C)$ and call $\phi_C := \phi_{\mathrm{out}(C)}$ the **Boolean formula corresponding to the circuit *C***. The **Boolean function *f*$_C$ realized by the circuit *C*** (also, **computed by *C***), is defined to be $f_C := f_{\phi_C}$.

Usually, a fixed basis Ω is considered, e.g., the basis consisting of exactly the elementary Boolean formulae defined in Definition 2.1.6.

Definition 2.2.3. A circuit C is **monotone** if each gate is labeled with a monotone Boolean formula. The basis $\Omega_{\mathrm{mon}} := \{\,\textsc{And}2, \textsc{Or}2\,\}$ is called the **standard monotone basis**. The basis $\Omega_{\mathrm{nmon}} := \{\,\textsc{And}2, \textsc{Or}2, \textsc{Not}\,\}$ is called the **standard non-monotone basis**. A circuit is called **binary** if each gate has at arity most 2.

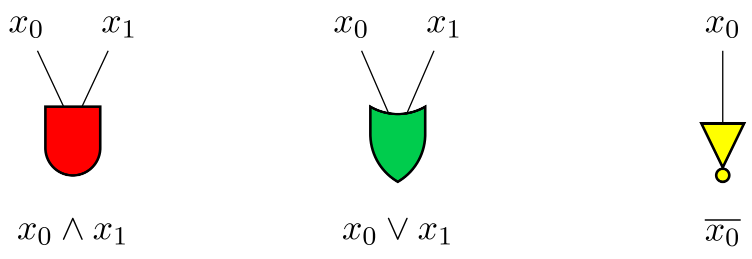
As every (monotone) Boolean function has a realizing (monotone) Boolean formula by Theorem 2.1.21 (Corollary 2.1.26) and every formula can be represented by a circuit, we obtain the following corollary.

Corollary 2.2.4. *For every (monotone) Boolean function $f\colon \{0,1\}^n \to \{0,1\}$, there is a (monotone) circuit over Ω_{nmon} (Ω_{mon}) realizing f.* □

A typical basis that is used for circuits on a computer chip is shown in Figure 7.1 (page 192).

Definition 2.2.5. Two circuits are called **equivalent** if they realize the same Boolean function.

When visualizing circuits, the three types of gates in the standard non-monotone basis are drawn as in Figure 2.2. The colors may vary in our pictures, but the shapes of the gates are fixed.



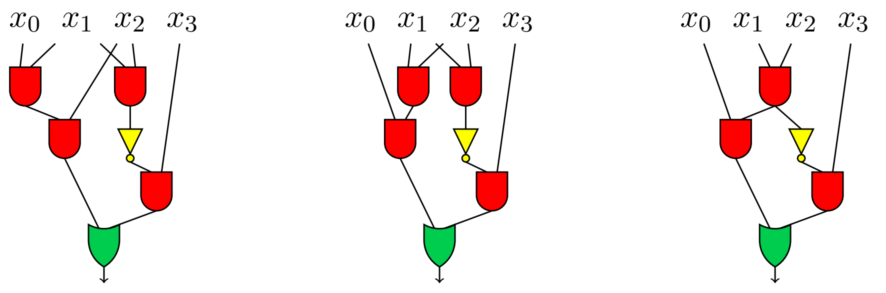
(a) An AND2 gate. **(b)** An OR2 gate. **(c)** A NOT gate.

Figure 2.2: Different types of gates.

Example 2.2.6. Figure Figure 2.3 depicts small circuits over Ω_{nmon} on the inputs x_0, x_1, x_2, x_3 with a single output each. The inputs with their associated variables are drawn at the top, and the directed graph is plotted from top to bottom (omitting edge directions) using the gate symbols from Figure 2.2. The circuit outputs are marked with an arrow; but when there are no outputs other than the vertices with out-degree 0, we may omit this arrow. We do not specify the ordering of gate inputs in the pictures when the gate is a Boolean formula whose associated Boolean function is invariant under permutation of the inputs.

The circuit C_1 shown in Figure 2.3(a) has the corresponding Boolean formula

$$\phi_1\big((x_0, x_1, x_2, x_3)\big) = \big((x_0 \wedge x_1) \wedge x_2\big) \vee \big(\overline{x_1 \wedge x_2} \wedge x_3\big)\,,$$



(a) Circuit C_1. **(b)** Circuit C_2. **(c)** Circuit C_3.

Figure 2.3: Three equivalent circuits C_1, C_2, and C_3 realizing the Boolean formulae ϕ_1, ϕ_2, and ϕ_3 from Example 2.2.6, respectively.

while the circuits C_2 and C_3 shown in Figures 2.3(b) and 2.3(c) both have the corresponding Boolean formula

$$\phi_2\big((x_0, x_1, x_2, x_3)\big) = \big(x_0 \wedge (x_1 \wedge x_2)\big) \vee \big(\overline{x_1 \wedge x_2} \wedge x_3\big) .$$

Still, C_2 and C_3 have a different number of gates since in C_3, the gate corresponding to $x_1 \wedge x_2$ has two successors while C_2 has two gates corresponding to $x_1 \wedge x_2$. As $\phi_1 = \phi_2$, all three circuits realize the same Boolean function.

Hence, each circuit is associated with a Boolean formula, but there can be different circuits with the same corresponding Boolean formula. In the restricted set of circuits where each gate has exactly 1 successor, each circuit corresponds uniquely to a Boolean formula.

Definition 2.2.7. A circuit with out-degree at most 1 for each gate vertex is called a **formula circuit**.

Observation 2.2.8. For each Boolean formula ϕ, there is a unique formula circuit C_ϕ over Ω_{nmon} corresponding to ϕ.

The next definition introduces naming conventions in the context of circuits.

Definition 2.2.9. Consider a circuit C and a vertex $v \in \mathcal{V}(C)$. The **fanout** of v is the number $\text{fanout}(v) := |\delta^+(v)|$ of outgoing edges of v. The set $\mathcal{V}_v(C) \subseteq \mathcal{V}(C)$ of all vertices $w \in \mathcal{V}(C)$ such that there is a directed path from w to v is called the **input cone** of v. By $\mathcal{I}_v(C)$, we denote the set of inputs in the input cone of v. The circuit C_v with $\mathcal{E}(C_v) = \mathcal{E}(C) \cap \big(\mathcal{V}(C_v) \times \mathcal{V}(C_v)\big)$, inputs $\mathcal{I}_v(C)$, gates $\mathcal{G}(C) \cap \mathcal{V}_v(C)$ and a single output v is called the **circuit subordinate to** v. Any circuit whose gates are a subset of $\mathcal{G}(C)$ is called a **sub-circuit** of C. If $v \in \mathcal{G}(C)$, then the **fanin** or **arity** of v is the number $|\delta^-(v)|$ of incoming edges of v.

The concept of duality (cf. Section 2.1.3) naturally extends from Boolean formulae to circuits.

Definition 2.2.10. Given a circuit C, the **dual circuit** C^* arises from C by interchanging all AND and OR gates and all 0 and 1 symbols.

Theorem 2.2.11. *For any circuit C, we have $\phi_{C^*} = {\phi_C}^*$ and $f_{C^*} = {f_C}^*$. Furthermore, for any Boolean formula ϕ, we have $C_{\phi^*} = C^*_\phi$.*

Proof. By Definitions 2.1.28 and 2.2.10, the duals of Boolean formula and circuits are both defined via exchanging And and Or operators (gates), thus $\phi_{C^*} = {\phi_C}^*$ and $C_{\phi^*} = C_\phi^*$ certainly hold. As ϕ_{C^*} realizes f_{C^*} and, by Theorem 2.1.31, ${\phi_C}^*$ realizes ${f_C}^*$, this implies $f_{C^*} = {f_C}^*$ by Corollary 2.1.32. □

2.3 Optimization Problems

Given a Boolean function f, by Theorem 2.1.21, there exists a Boolean formula realizing f, and by Corollary 2.2.4, there exists a circuit realizing f. But often, there are multiple Boolean formulae and circuits realizing f, see, e.g., Example 2.2.6. Thus, finding the best formula or the best circuit realizing f regarding a certain objective function is a natural problem. We introduce different quality measures in Section 2.3.1 and examine them in the subsequent sections.

2.3.1 Quality Measures

The application of circuits in chip design yields several quality measures.

Definition 2.3.1. Consider a circuit C. The **depth** of a vertex $v \in \mathcal{V}(C)$ is

$$\operatorname{depth}(v) := \max_{P \text{ directed path in } C \text{ ending in } v} |E(P)| \,.$$

The **depth** of C is the maximum depth of any vertex $v \in \mathcal{V}(C)$.

In other words, the depth of a circuit C is the maximum number of gates on any directed path from an input of C to an output of C.

If a circuit models a part of computer chip, the input variables of the circuit represent signals computed by other circuits. As these signals are not necessarily available simultaneously, we can generalize Definition 2.3.1 as follows.

Definition 2.3.2. Consider a circuit C on inputs $x = (x_0, \dots, x_{n-1})$. Assume that input **arrival times** $a(x_0), \dots, a(x_{n-1}) \in \mathbb{R}$ are given, i.e., input x_i has arrival time $a(x_i) \in \mathbb{R}$ for each $i \in \{0, \dots, n-1\}$. Recursively, we define arrival times $a(v) \in \mathbb{R}$ for all $v \in \mathcal{G}(C)$ by setting

$$a(v) := \max_{w \in \delta^-(v)} \{ a(w) \} + 1 \,.$$

Moreover, we define the **delay** of C with respect to arrival times a as

$$\operatorname{delay}(C; a) := \max_{v \in \mathcal{O}(C)} a(v) \,.$$

When the arrival times a can be deduced from the context, we also write $\operatorname{delay}(C) := \operatorname{delay}(C; a)$.

Definition 2.3.1 covers the special case of Definition 2.3.2 when the input arrival times are all 0.

The delay of a circuit C models the time when the function at the outputs of C is available given that the input signals arrive at prescribed times. In our practical application in chip design, a weakness of our delay model is that it ignores the fact that signals slow down when they are distributed too often, see Section 7.1. Thus, we need to take care of the following measuring unit.

Definition 2.3.3. Given a circuit C, the **fanout** $\operatorname{fanout}(C)$ of C is the maximum fanout of any vertex in C.

Besides speed, the size, power consumption and production cost of a chip are important factors. These are estimated by the following metric.

Definition 2.3.4. Given a circuit C, the **size** of C is the number of gates of C.

Observation 2.3.5. Given a circuit C with a single output that contains only gates with fanin at most 2, we have $\operatorname{size}(C) \leq 2^{\operatorname{depth}(C)} - 1$.

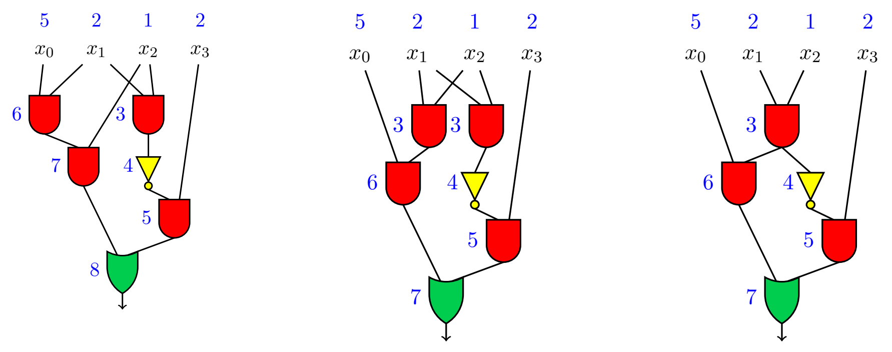
For a formula circuit, size and fanout are closely related.

Observation 2.3.6. Let C be a formula circuit over a basis Ω containing only formulae with arity exactly 2. Then, we have

$$2|\mathcal{G}(C)| = |\mathcal{E}(C)| = \sum_{v \in \mathcal{I}(C)} \operatorname{fanout}(v) + |\mathcal{G}(C)| - |\mathcal{O}(C)|\,.$$

This implies $\operatorname{size}(C) = |\mathcal{G}(C)| = \sum_{v \in \mathcal{I}(C)} \operatorname{fanout}(v) - |\mathcal{O}(C)|$.

Example 2.3.7. In Figure 2.4, we plot again the equivalent circuits from Figure 2.3, but with certain arrival times drawn in blue. We have $\operatorname{depth}(C_i) = 4$ and $\operatorname{fanout}(C_i) = 2$ for all $i \in \{1,2,3\}$, but $\operatorname{delay}(C_1) = 8$, $\operatorname{delay}(C_2) = \operatorname{delay}(C_3) = 7$.



(a) Circuit C_1. **(b)** Circuit C_2. **(c)** Circuit C_3.

Figure 2.4: Three circuits realizing the function f from Example 2.3.7. The blue numbers refer to arrival times.

We can define similar properties of Boolean formulae via their associated circuits.

Definition 2.3.8. Consider a Boolean formula ϕ with inputs $x = (x_0, \ldots, x_{n-1})$ and the associated formula circuit C_ϕ. We define the **depth** (**size**) of ϕ to be the depth (size) of C_ϕ. Assuming that input x_i has arrival time $a(x_i) \in \mathbb{N}$ for each $i \in \{0, \ldots, n-1\}$, we define the **delay** of ϕ to be the delay of C_ϕ.

Using this definition and duality (see Theorem 2.2.11), it is easy to see that the introduced quality measures are invariant under dualization.

Proposition 2.3.9. *Given a circuit C on Boolean inputs $x_0, \ldots, x_{n-1}$ with input arrival times $a(x_0), \ldots, a(x_{n-1}) \in \mathbb{R}$, we have* $\operatorname{size}(C^*) = \operatorname{size}(C)$, $\operatorname{depth}(C^*) = \operatorname{depth}(C)$, *and* $\operatorname{delay}(C^*; a) = \operatorname{delay}(C; a)$. *Given a Boolean formula ϕ, we have* $\operatorname{size}(\phi^*) = \operatorname{size}(\phi)$, $\operatorname{depth}(\phi^*) = \operatorname{depth}(\phi)$, *and* $\operatorname{delay}(\phi^*; a) = \operatorname{delay}(\phi; a)$. $\square$

2.3.2 Delay Optimization

In this thesis, we focus on the delay optimization problems for certain classes of Boolean functions.

Boolean Formula Delay Optimization Problem

Instance:	A Boolean function $f\colon \{0,1\}^n \to \{0,1\}$ on inputs $x = (x_0, \dots, x_{n-1})$, input arrival times $a(x_0), \dots, a(x_{n-1}) \in \mathbb{R}$.
Task:	Find a Boolean formula realizing $f(x)$ with minimum possible delay.

Circuit Delay Optimization Problem

Instance:	A Boolean function $f\colon \{0,1\}^n \to \{0,1\}$ on inputs $x = (x_0, \dots, x_{n-1})$, arrival times $a(x_0), \dots, a(x_{n-1}) \in \mathbb{R}$, and a basis Ω.
Task:	Compute a circuit over Ω realizing $f(x)$ with minimum possible delay.

Remark 2.3.10. In general, when the input function is given by a Boolean formula, the above two problems are NP-hard already for the special case of depth optimization. To see this, we show that a polynomial-time algorithm finding depth-optimum Boolean formulae for And-Or paths could be used to solve the Satisfiability Problem in polynomial time:

Given a Boolean formula $\phi(x)$ with Boolean input variables $x = (x_0, \dots, x_{n-1})$, the Satisfiability Problem asks whether there is an $\alpha \in \{0,1\}^n$ with $\phi(\alpha) = 1$. I.e., ϕ is not satisfiable if and only if ϕ evaluates to 0 for all $\alpha \in \{0,1\}^n$. Hence, ϕ is satisfiable if and only if a depth-minimum equivalent formula ψ for ϕ fulfills $\psi \neq 0$.

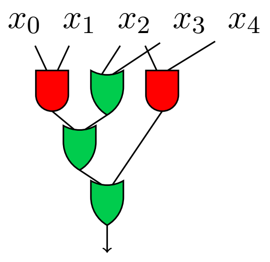
In this thesis, we consider this problem only for certain types of Boolean functions.

As the delay of a Boolean formula is defined as the delay of its unique corresponding circuit (see Definition 2.3.8), the Boolean Formula Delay Optimization Problem is a special case of the Circuit Delay Optimization Problem with basis $\Omega := \Omega_{\text{nmon}}$ where only formula circuits are considered. Sometimes, we restrict the solution space for either problem to monotone solutions. As our research interest is motivated by chip design, we usually focus on the Circuit Delay Optimization Problem, but sometimes, we also explicitly make use of the fact that our solutions are formula circuits. If there are no constraints on size and fanout, the Boolean Formula Delay Optimization Problem and the Circuit Delay Optimization Problem are clearly equivalent:

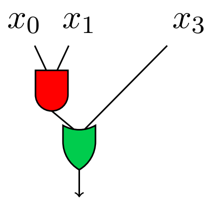
Theorem 2.3.11. *Let a Boolean function $f\colon \{0,1\}^n \to \{0,1\}$ on input variables $x_0, \dots, x_{n-1}$ with arrival times $a(x_0), \dots, a(x_{n-1}) \in \mathbb{R}$ be given. Fix a basis $\Omega \in \{\Omega_{\text{mon}}, \Omega_{\text{nmon}}\}$. For this instance, the optimum delays of any two solutions of the* Boolean Formula Delay Optimization Problem *and* Circuit Delay Optimization Problem *coincide.*

Proof. Consider optimum solutions ϕ_{opt} and C_{opt} for the Boolean Formula Delay Optimization Problem and Circuit Delay Optimization Problem, respectively. As any Boolean formula yields a formula circuit with the same delay by Definition 2.3.8, we have $\text{delay}(\phi_{\text{opt}}) = \text{delay}(C_{\phi_{\text{opt}}}) \geq \text{delay}(C_{\text{opt}})$.

For the reverse statement, it suffices to apply the following claim to C_{opt}.



(a) A circuit C on 5 inputs.



(b) Reduced circuit $C\mid_{x_2=0}$.

Figure 2.5: Illustration of Observation 2.3.14. Note that here, $C\mid_{x_2=0}$ does not depend essentially on x_4.

Claim. For any circuit C over Ω, there is an equivalent formula circuit F over Ω with $\operatorname{delay}(F) = \operatorname{delay}(C)$.

Proof of claim: We call a gate g in C with $\operatorname{fanout}(g) > 1$ **bad**. Let $d \in \mathbb{N}$ be the largest depth of any bad gate in C. We prove the statement by induction on d.

If $d = 0$, then the circuit C is already a formula circuit.

If $d > 0$, consider a bad gate g with depth exactly d. Denote the successors of g by $v_0, \ldots, v_{f-1}$, where $f \geq 2$. Replace g by f copies $g_0, \ldots, g_{f-1}$, where each g_i, $i = 0, \ldots, f-1$, has the same predecessors as g, but only has v_i as a successor. Denote the circuit arising from C by applying this to every bad gate with depth d by C'. Then, C' is a circuit over Ω that is equivalent to C, we have $\operatorname{delay}(C') = \operatorname{delay}(C)$, and the largest depth of any bad gate is at most $d-1$. By induction hypothesis, there is a formula circuit F equivalent to C' and thus to C with $\operatorname{delay}(F) = \operatorname{delay}(C') = \operatorname{delay}(C)$. □

□

Note that the circuit in Figure 2.4(b) arises from the circuit in Figure 2.4(c) by the procedure described in the proof of Theorem 2.3.11. In this case, the number of gates increases by 1 and the number of edges by 2.

Note that in the claim in Theorem 2.3.11, the circuits C and C' have the same corresponding Boolean formula. This implies the following statement.

Corollary 2.3.12. *For any circuit C over Ω_{nmon} or Ω_{mon}, the Boolean formula ϕ_C corresponding to C fulfills* $\operatorname{delay}(\phi_C) = \operatorname{delay}(C)$. □

In the theoretical chapters of this thesis, we will often assume that the input arrival times are integral. In this case, if additionally the Boolean formulae in the basis have all arity r or less, there is a lower bound by Golumbic [Gol76] on the optimum solution of the CIRCUIT DELAY OPTIMIZATION PROBLEM stated in Theorem 2.3.15. For this, we need to introduce the concept of reduced circuits, for which an example is shown in Figure 2.5.

Definition 2.3.13. Consider a monotone Boolean function $f\colon \{0,1\}^n \to \{0,1\}$ on input variables $x_0, \ldots, x_{n-1}$, an index $i \in \{0, \ldots, n-1\}$ and a value $\alpha \in \{0,1\}$. Consider a circuit C for f over Ω_{mon}. The **reduced circuit** $C\mid_{x_i=\alpha}$ arises from C as follows: Replace x_i by α and apply the following to each gate $g \in \mathcal{G}(C)$ in topological order:

Assume that there is a predecessor $v \in \delta^-(g)$ which is a constant (otherwise, do nothing for g), and denote the other predecessor of g by w.

Case 1: Assume that $\text{gt}(g) = \text{And}$ and $v = 0$, or $\text{gt}(g) = \text{Or}$ and $v = 1$. Replace each edge $(g, y) \in \delta^+(g)$ by (v, y). If $g \in \mathcal{O}(C)$, then let $\mathcal{O}(C) := \big(\mathcal{O}(C) \backslash \{g\}\big) \cup \{v\}$.

Case 2: Otherwise, replace each edge $(g, y) \in \delta^+(g)$ by (w, y), and if $g \in \mathcal{O}(C)$, then let $\mathcal{O}(C) := \big(\mathcal{O}(C) \backslash \{g\}\big) \cup \{w\}$.

Remove g from $\mathcal{V}(C)$.

Observation 2.3.14. Consider a Boolean function $f \colon \{0,1\}^n \to \{0,1\}$ on input variables $x_0, \ldots, x_{n-1}$ with arrival times $a(x_0), \ldots, a(x_{n-1}) \in \mathbb{R}$, an index $i \in \{0, \ldots, n-1\}$ and a value $\alpha \in \{0,1\}$. Consider a circuit C for f and the reduced circuit $C\ |_{x_i=\alpha}$. Then, $C\ |_{x_i=\alpha}$ is a circuit for the restricted function $f\ |_{x_i=\alpha}$. Moreover, we have $\text{delay}(C\ |_{x_i=\alpha}) \leq \text{delay}(C)$ and $\text{size}(C\ |_{x_i=\alpha}) \leq \text{size}(C)$. If $\text{fanout}(x_i) > 0$, then we have $\text{size}(C\ |_{x_i=\alpha}) < \text{size}(C)$.

Theorem 2.3.15 (Cf. Golumbic [Gol76])**.** *Consider a Boolean function $f \colon \{0,1\}^n \to \{0,1\}$ on inputs $x_0, \ldots, x_{n-1}$ with input arrival times $a(x_0), \ldots, a(x_{n-1}) \in \mathbb{N}$. Assume that f depends essentially on all its inputs. Furthermore, let $r \in \mathbb{N}$, $r \geq 2$, and a basis Ω containing only Boolean formulae of arity at most r be given. Then, for any circuit C over Ω computing f, we have*

$$\text{delay}(C) \geq \left\lceil \log_r \left(\sum_{i=0}^{n-1} r^{a(x_i)} \right) \right\rceil .$$

Proof. We prove the statement by induction on $d := \text{depth}(C)$.

If $d = 0$, we have $n = 1$ and the delay of C is $a(x_0)$ which is also the right-hand side of the claimed inequality.

For $d \geq 1$, let $g := \text{out}(C)$ be the unique output of C. Denote the predecessors of g by $v_0, \ldots, v_{k-1}$ with $k \leq r$. For each $j \in \{0, \ldots, k-1\}$, let $I_j \subseteq \mathcal{I}(C)$ denote the inputs that C_{v_j} depends on essentially. Let C_j denote the reduced circuit arising from C_{v_j} by fixing each $x_i \in \mathcal{I}(C) \backslash I_j$ to false. As C depends essentially on all its inputs, we have $\mathcal{I} = \bigcup_j I_j$. Since $\text{depth}(C_j) \overset{\text{Obs. 2.3.14}}{\leq} \text{depth}(C_{v_j}) < d$ for all j, the induction hypothesis holds for all C_j. Choose $j^* \in \{0, \ldots, k-1\}$ such that

$\sum_{x_i \in C_{j^*}} r^{a(x_i)}$ is maximum. We have

$$\begin{aligned}
\text{delay}(C) &= \max_{j \in \{0,\dots,k-1\}} \text{delay}(C_{v_j}) + 1 \\
&\overset{\text{Obs. 2.3.14}}{\geq} \max_{j \in \{0,\dots,k-1\}} \text{delay}(C_j) + 1 \\
&\overset{\text{(IH)}}{\geq} \max_{j \in \{0,\dots,k-1\}} \left\lceil \log_r \left(\sum_{x_i \in C_j} r^{a(x_i)} \right) \right\rceil + 1 \\
&\overset{\text{choice of } j^*}{=} \left\lceil \log_r \left(\sum_{x_i \in C_{j^*}} r^{a(x_i)} \right) \right\rceil + 1 \\
&\geq \log_r \left(r \sum_{x_i \in C_{j^*}} r^{a(x_i)} \right) \\
&\overset{\substack{\text{choice of } j^*, \\ k \leq r}}{\geq} \log_r \left(\sum_{j=0}^{k-1} \sum_{x_i \in C_j} r^{a(x_i)} \right) \\
&\overset{\mathcal{I} = \bigcup_i I_j}{\geq} \log_r \left(\sum_{i=0}^{n-1} r^{a(x_i)} \right).
\end{aligned}$$

Since $\text{delay}(C) \in \mathbb{N}$, we conclude $\text{delay}(C) \geq \left\lceil \log_r \left(\sum_{i=0}^{n-1} r^{a(x_i)} \right) \right\rceil$. □

Werber [Wer07] states an alternative proof that reduces the lower bound to Kraft's inequality [Kra49], see also Proposition 2.3.22. For the special case of depth optimization, Theorem 2.3.15 implies

$$\text{depth}(C) \geq \lceil \log_r n \rceil$$

which was proven by Winograd [Win65].

We mostly consider binary circuits and hence choose $r = 2$ in the preceding theorem. For this case, the following definition will be convenient.

Definition 2.3.16. The **weight** of inputs $x = (x_0, \dots, x_{n-1})$ with arrival times $a(x_0), \dots, a(x_{n-1}) \in \mathbb{N}$ is

$$W(x; a) := \sum_{i=0}^{n-1} 2^{a(x_i)}.$$

When the arrival times can be derived from the context, we abbreviatory write $W(x) := W(x; a)$.

Remark 2.3.17. Abusing notation, we write $W(x_i) := 2^{a(x_i)}$ for the weight of an input x_i with arrival time $a(x_i)$. Note that if $a(x_i) \geq 0$, we have $W(x_i) \geq 1$.

2.3.3 Symmetric Function Optimization

A class of functions for which the BOOLEAN FORMULA DELAY OPTIMIZATION PROBLEM and the CIRCUIT DELAY OPTIMIZATION PROBLEM coincide and can be solved optimally and efficiently is defined as follows.

Algorithm 2.1: Delay optimization for symmetric functions

```
Input: A commutative and associative operation
       $\circ\colon \{0,1\} \times \{0,1\} \to \{0,1\}$, $n \in \mathbb{N}$, $n \geq 1$, inputs $x = (x_0, \dots, x_{n-1})$
       with arrival times $a(x_0), \dots, a(x_{n-1}) \in \mathbb{R}$.
Output: A circuit $C$ over the basis $\Omega = \{\circ\}$ computing $x_0 \circ \dots \circ x_{n-1}$.
1 Let $x_0, \dots, x_{n-1}$ be the inputs of $C$.
2 Let $Q := \{v \in \mathcal{V}(C) : \delta^+(v) = \emptyset\}$.
  // Define gates.
3 while $|Q| \geq 2$ do
4     Choose $v, w \in Q$, $v \neq w$, with $a(v), a(w)$ minimum.
5     Add a new $\circ$-gate $g$ with predecessors $v$ and $w$ to $C$.
6     Set $Q := Q \setminus \{v, w\} \cup \{g\}$.
7 Let out$(C)$ be the unique vertex $v \in Q$.
8 return $C$.
```

Definition 2.3.18. Let $\circ\colon \{0,1\} \times \{0,1\} \to \{0,1\}$ denote a commutative and associative operation. For any $n \in \mathbb{N}$, the Boolean function $f\colon \{0,1\}^n \to \{0,1\}$, $f\big((x_0, \dots, x_{n-1})\big) = x_0 \circ \dots \circ x_{n-1}$ is called **symmetric.** A circuit realizing a symmetric function is also called **symmetric**.

SYMMETRIC FUNCTION DELAY OPTIMIZATION PROBLEM

Instance: $n \in \mathbb{N}$, Boolean input variables $x = (x_0, \dots, x_{n-1})$ with arrival times $a(x_0), \dots, a(x_{n-1}) \in \mathbb{N}$, a commutative and associative operation $\circ\colon \{0,1\} \times \{0,1\} \to \{0,1\}$.

Task: Compute a circuit over $\Omega = \{\circ\}$ realizing $x_0 \circ \dots \circ x_{n-1}$ with minimum possible delay.

Example 2.3.19. Since $\wedge$ and $\vee$ are commutative and associative by Proposition 2.1.10, the n-ary AND and OR functions are symmetric. Furthermore, the n-ary XOR function is a symmetric function since $\oplus$ is obviously commutative, but also associative since $x \oplus (y \oplus z)$ is true if and only if an odd number of the variables x, y, z is true.

Definition 2.3.20. Given input variables $x_0, \dots, x_{n-1}$, we denote the n-ary AND function by $\operatorname{sym}\big((x_0, \dots, x_{n-1})\big) = \bigwedge_{i=0}^{n-1} x_i$ and its dual function – the n-ary OR function – by $\operatorname{sym}^*\big((x_0, \dots, x_{n-1})\big) = \bigvee_{i=0}^{n-1} x_i$.

Algorithm 2.1, which is a variant of Huffman coding [Huf52], computes a solution to the SYMMETRIC FUNCTION DELAY OPTIMIZATION PROBLEM. By Theorem 2.3.15, for integral arrival times, this circuit has delay at least $\left\lceil \log_2\big(W(x;a)\big) \right\rceil$. The following theorem states that this delay is actually attained.

Theorem 2.3.21 (Golumbic [Gol76] and Van Leeuwen [Lee76])**.** *Let a commutative and associative operation* $\circ\colon \{0,1\} \times \{0,1\} \to \{0,1\}$*, inputs* $x_0, \dots, x_{n-1}$ *with* $n \geq 1$ *and arrival times* $a(x_0), \dots, a(x_{n-1}) \in \mathbb{N}$ *be given. Algorithm 2.1 computes an optimum solution* C *to the* SYMMETRIC FUNCTION DELAY OPTIMIZATION PROBLEM *with*

$$\operatorname{delay}(C) = \left\lceil \log_2\big(W(x;a)\big) \right\rceil .$$

If we assume that the inputs are sorted by increasing arrival time, then Algorithm 2.1 can be implemented to run in time $\mathcal{O}(n)$; otherwise, in time $\mathcal{O}(n \log_2 n)$.

The proof of the delay bound is due to Golumbic [Gol76] (see Werber [Wer07] for a concise proof), while Van Leeuwen [Lee76] showed that the algorithm can be implemented to run in linear time after sorting.

Theorem 2.3.21 is closely related to Kraft's inequality, which was stated in a very different form by Kraft [Kra49] and can be re-proven easily now.

Proposition 2.3.22 (Kraft's inequality, Kraft [Kra49])**.** *Given a commutative and associative operation $\circ\colon \{0,1\} \times \{0,1\} \to \{0,1\}$, inputs $x_0, \dots, x_{n-1}$ with $n \geq 1$ and arrival times $a(x_0), \dots, a(x_{n-1}) \in \mathbb{N}$, there exists a circuit S computing $x_0 \circ \dots \circ x_{n-1}$ with delay at most $d \in \mathbb{N}$ if and only if*

$$\sum_{i=0}^{n-1} 2^{a(x_i)-d} \leq 1\,. \tag{2.14}$$

Proof. By Theorem 2.3.21, a circuit S for $x_0 \circ \dots \circ x_{n-1}$ with delay at most $d \in \mathbb{N}$ exists if and only if $\left\lceil \log_2\bigl(W(x;a)\bigr) \right\rceil \leq d$. As d is an integer, this condition is equivalent to $\log_2\bigl(W(x;a)\bigr) \leq d$, thus to $\sum_{i=0}^{n-1} 2^{a(x_i)} \leq 2^d$, which is again equivalent to condition (2.14). □

We shall now generalize Kraft's inequality and Huffman coding to fractional arrival times using standard techniques (cf. Bartoschek et al. [Bar+10]).

Proposition 2.3.23. *Let a commutative, associative operation $\circ\colon \{0,1\} \times \{0,1\} \to \{0,1\}$ and inputs $x_0, \dots, x_{n-1}$ for $n \geq 1$ with arrival times $a(x_0), \dots, a(x_{n-1}) \in \mathbb{R}$ be given. There exists a circuit S computing $x_0 \circ \dots \circ x_{n-1}$ with delay $d \in \mathbb{R}$ if and only if*

$$\sum_{i=0}^{n-1} 2^{-\lceil d - a(x_i) \rceil} \leq 1\,. \tag{2.15}$$

Proof. First, we prove the following claim:

Claim. Proposition 2.3.22 still holds if $a\colon \{x_0, \dots, x_{n-1}\} \to \mathbb{Z}$.

Proof of claim: Define modified arrival times $\tilde{a}(x_i) := a(x_i) - \alpha$, where $\alpha := \min_{i \in \{0,\dots,n-1\}}\{a(x_i)\}$. Now, $\tilde{a}(x_i) \geq 0$ for all $i \in \{0, \dots, n-1\}$, and a circuit S on inputs $x_0, \dots, x_{n-1}$ has delay d with respect to arrival times a if and only if it has delay $\tilde{d} := d - \alpha$ with respect to arrival times $\tilde{a}$. By Proposition 2.3.22, this is equivalent to $\sum_{i=0}^{n-1} 2^{\tilde{a}(x_i)-\tilde{d}} \leq 1$ and hence, by definition of $\tilde{a}$ and $\tilde{d}$, to $\sum_{i=0}^{n-1} 2^{a(x_i)-d} \leq 1$. □

Now consider arbitrary fractional arrival times $a\colon \{x_0, \dots, x_{n-1}\} \to \mathbb{R}$. A circuit S has delay $d \in \mathbb{R}$ if and only if for any directed path P from any input x_i to $\text{out}(S)$, we have $a(x_i) + |P| \leq d$, i.e., if and only if

$$\max_{i \in \{0,\dots,n-1\}, P\colon x_i \rightsquigarrow \text{out}(S)} \bigl\{ |P| - (d - a(x_i)) \bigr\} \leq 0\,.$$

Since $|P|$ and 0 are integers, this is equivalent to

$$\max_{i \in \{0,\dots,n-1\}, P\colon x_i \rightsquigarrow \text{out}(S)} \Bigl\{ |P| - \lceil d - a(x_i) \rceil \Bigr\} \leq 0\,.$$

In other words, this means that for modified arrival times $\tilde{a}(x_i) := -\lceil d - a(x_i) \rceil \in \mathbb{Z}$, the circuit S has delay 0. By the claim, this is equivalent to condition (2.15). □

Hence, the delay of an optimum symmetric circuit on inputs $x_0, \dots, x_{n-1}$ with fractional arrival times $a(x_0), \dots, a(x_{n-1}) \in \mathbb{R}$ is the minimum value $d \in \mathbb{R}$ such that

$$\sum_{i=0}^{m-1} 2^{-\lceil d - a(t_i) \rceil} \leq 1 .$$

Recall from Theorem 2.3.21 that the optimum delay of a symmetric tree S on inputs $x_0, \dots, x_{n-1}$ with integral arrival times $a(x_0), \dots, a(x_{n-1})$ can be read off from the weight of the inputs: $\text{delay}(S) = \left\lceil \log_2\left(\sum_{i=0}^{n-1} 2^{a(x_i)}\right) \right\rceil$. However, for fractional arrival times, this is not the case.

Example 2.3.24. Let $\varepsilon > 0$ and $k, l \in \mathbb{N}$ with $k = 2^l$ and k and l sufficiently large such that $k2^\varepsilon - 1 \geq k$. Consider two instances for the Symmetric Function Delay Optimization Problem: Let Boolean input variables $x = (x_0, \dots, x_{k-1})$ with arrival times $a(x_i) = \varepsilon$ for all $i \in \{0, \dots, k-1\}$ and Boolean input variables $y = (y_0, y_1)$ with arrival times $a(y_0) = 0$, $a(y_1) = \log_2(k2^\varepsilon - 1)$ be given. Then, we have $\sum_{i=0}^{k-1} 2^{a(x_i)} = k2^\varepsilon$ and $\sum_{i=0}^{1} 2^{a(y_i)} = 1 + (k2^\varepsilon - 1) = k2^\varepsilon$. A full binary tree on the inputs of x has delay $\log_2 k + \varepsilon = l + \varepsilon$, while an optimum symmetric tree on y is $y_0 \circ y_1$, which has delay $\log_2(k2^\varepsilon - 1) + 1 \overset{k2^\varepsilon - 1 \geq k}{\geq} \log_2 k + 1 = l + 1$.

Still, we can estimate the delay of a symmetric tree for inputs with fractional arrival times up to an additive error of 1:

Observation 2.3.25. Let Boolean input variables $x = (x_0, \dots, x_{n-1})$ with fractional arrival times $a(x_0), \dots, a(x_{n-1}) \in \mathbb{R}$ be given. Consider an optimum symmetric tree S on inputs x. Define arrival times $a_l(x_i) = \lfloor a(x_i) \rfloor$ and $a_u(x_i) = \lceil a(x_i) \rceil$ for all $i \in \{0, \dots, n-1\}$. Note that $\text{delay}(S; a_l) \leq \text{delay}(S; a) \leq \text{delay}(S; a_u)$. By Theorem 2.3.21, we have $\text{delay}(S; a_l) = \left\lceil \log_2\left(\sum_{i=0} 2^{\lfloor a(x_i) \rfloor}\right) \right\rceil$ and $\text{delay}(S; a_u) = \left\lceil \log_2\left(\sum_{i=0} 2^{\lceil a(x_i) \rceil}\right) \right\rceil \leq \left\lceil \log_2\left(\sum_{i=0} 2^{\lfloor a(x_i) \rfloor}\right) \right\rceil + 1$. Together, this implies

$$\left\lceil \log_2\left(\sum_{i=0} 2^{\lfloor a(x_i) \rfloor}\right) \right\rceil \leq \text{delay}(S; a) \leq \left\lceil \log_2\left(\sum_{i=0} 2^{\lfloor a(x_i) \rfloor}\right) \right\rceil + 1 .$$

However, Huffman coding computes an optimum solution also for fractional arrival times:

Proposition 2.3.26. *Given a commutative and associative operation $\circ\colon \{0,1\} \times \{0,1\} \to \{0,1\}$, input variables $x_0, \dots, x_{n-1}$ with $n \geq 1$ and arrival times $a(x_0), \dots, a(x_{n-1}) \in \mathbb{R}$, Huffman coding (Algorithm 2.1) computes an optimum solution to the* Circuit Delay Optimization Problem *for symmetric functions over $\{\circ\}$. If we assume that the inputs are sorted by increasing arrival time, then Algorithm 2.1 can be implemented to run in time $\mathcal{O}(n \log_2 n)$.*

Proof. By the proof of Proposition 2.3.23, an optimum symmetric circuit for inputs $x_0, \dots, x_{n-1}$ and arrival times $a\colon \{x_0, \dots, x_{n-1}\} \to \mathbb{R}$ can be computed using Huffman coding (Algorithm 2.1) on auxiliary arrival times $\tilde{a}(x_i) := -\lceil d - a(x_i) \rceil \in \mathbb{Z}$ for $i \in \{0, \dots, n-1\}$ once the optimum delay $d \in \mathbb{R}$ is known. But note that for choosing the vertices v and w in line 4 with respect to $\tilde{a}$, the vertices v and w with minimum $a(v), a(w)$ are always a valid choice. Hence, Huffman coding with respect to the original arrival times also computes an optimum solution.

It is not clear whether Van Leeuwen's linear-time algorithm from [Lee76] for integral arrival times can be extended to the fractional case. However, using a heap, Algorithm 2.1 can be implemented to run in $\mathcal{O}(n \log_2 n)$. □

2.4 Adder Circuits

We now formally introduce binary addition and the problem of finding fast circuits for binary addition, which is one of the main topics considered in this work.

Definition 2.4.1. Let $n \in \mathbb{N}$. The **summation function** for n-bit binary numbers is defined as $s_n \colon \{0,1\}^{2n} \to \{0,1\}^{n+1}$, $s_n\big((a,b)\big) = a + b$, where we view $a = (a_0, \dots, a_{n-1})$ and $b = (b_0, \dots, b_{n-1})$ as two n-bit binary numbers with most significant bit $n-1$.

BASIC ADDER OPTIMIZATION PROBLEM

Instance: $n \in \mathbb{N}$.

Task: Compute a circuit over $\Omega = \{\text{AND}, \text{OR}, \text{NOT}, \text{XOR}\}$ realizing the summation function $s_n \colon \{0,1\}^{2n} \to \{0,1\}^{n+1}$.

In literature, a solution to this problem is usually constructed via carry bits, see, e.g., Weinberger and Smith [WS58] and Knowles [Kno99]:

Definition 2.4.2. Consider two binary numbers $a = (a_0, \dots, a_{n-1})$ and $b = (b_0, \dots, b_{n-1})$ with most significant bit $n-1$. For $i \in \{0, \dots, n-1\}$, we call $g_i := a_i \wedge b_i \in \{0,1\}$ the i-th **generate signal** and $p_i := a_i \oplus b_i \in \{0,1\}$ the i-th **propagate signal** for a and b. Recursively, we define the **carry bits** $c_0, \dots, c_n \in \{0,1\}$:

$$\begin{aligned} c_0 &= 0 \\ c_{i+1} &= g_i \vee (p_i \wedge c_i) \text{ for } 0 \leq i \leq n-1 \end{aligned} \tag{2.16}$$

Said in words, once the carry bit c_i is computed, we can determine the carry bit c_{i+1} since this is true if and only if

- c_{i+1} is *generated* at position i since a_i and b_i are both true, i.e., $g_i = a_i \wedge b_i$ is true, or
- c_{i+1} is *propagated* from position $i-1$ since c_i is true and exactly one of a_i and b_i is true, i.e., $p_i = a_i \oplus b_i$ is true.

From the carry bits, we can easily read off the sum:

Observation 2.4.3. Given two binary numbers $a = (a_0, \dots, a_{n-1})$ and $b = (b_0, \dots, b_{n-1})$ with most significant bit $n-1$, propagate signals $p_0, \dots, p_{n-1}$, and carry bits $c_0, \dots, c_n$, we have

$$\Big(s_n\big((a,b)\big)\Big)_i = \begin{cases} c_i \oplus p_i & \text{if } i \in \{0, \dots, n-1\}, \\ c_n & \text{if } i = n. \end{cases} \tag{2.17}$$

Since computing the propagate and generate signals as in Definition 2.4.2 and computing the sum from the propagate signals and carry bits as in (2.17) requires only a constant depth and linear size, most researchers including ourselves solve the following problem instead of the BASIC ADDER OPTIMIZATION PROBLEM.

ADDER OPTIMIZATION PROBLEM
Instance: $n \in \mathbb{N}$
Task: Construct a circuit over $\Omega_{\text{mon}} = \{\text{AND2}, \text{OR2}\}$ on n input pairs $p_0, g_0, \dots, p_{n-1}, g_{n-1}$ computing all the carry bits $c_1, \dots, c_n$.

Definition 2.4.4. A circuit solving the ADDER OPTIMIZATION PROBLEM for some $n \in \mathbb{N}$ is called an **adder circuit** or, short, **adder**. A family of circuits $(A_n)_{n \in \mathbb{N}_{>0}}$ where circuit A_n solves the ADDER OPTIMIZATION PROBLEM on n input pairs $p_0, g_0, \dots, p_{n-1}, g_{n-1}$ is called a **family of adder circuits**. Given $n \in \mathbb{N}$, $0 \in \{1, \dots, n\}$, and an adder circuit A_n, we denote the output of A_n computing the carry bit c_i by $\text{out}_i(A_n)$.

By Equation (2.16), an adder circuit on n input pairs $p_0, g_0, \dots, p_{n-1}, g_{n-1}$ does not depend essentially on p_0. However, to simplify notation, we mention p_0 as an input in Definition 2.4.4.

Remark 2.4.5. By Equation (2.16), the carry-bit function is monotone. Thus, by Corollary 2.1.26, there is always a monotone formula for each carry bit, and hence a circuit over $\Omega_{\text{mon}} = \{\text{AND}, \text{OR}\}$ solving the ADDER OPTIMIZATION PROBLEM. On the contrary, the summation function s_n is non-monotone: E.g., for $n = 1$, $a = (1)$, $b_0 = (0)$ and $b_1 = (1)$ with $b_0 < b_1$, we have $s_1\big((a, b_0)\big) = (0, 1) \not< (1, 0) = s_1\big((a, b_1)\big)$. Hence, a circuit solving the BASIC ADDER OPTIMIZATION PROBLEM is always non-monotone. However, it is an open question whether – beyond the computation of the propagate signals and the final sum – inverters can help to construct adder circuits with, say, a good depth (see also Section 2.6.1).

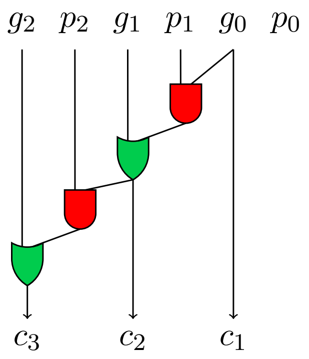
Expanding the recursive carry-bit definition (2.16) for a few steps, we obtain

$$\begin{aligned} c_{i+1} &= g_i \lor (p_i \land c_i) \\ &= g_i \lor \Big(p_i \land \big(g_{i-1} \lor (p_{i-1} \land c_{i-1})\big)\Big) \\ &= g_i \lor \Bigg(p_i \land \Bigg(g_{i-1} \lor \Big(p_{i-1} \land \Big(g_{i-2} \lor \big(p_{i-2} \land \dots (p_1 \land g_0)\big)\Big)\Big)\Bigg)\Bigg). \end{aligned} \tag{2.18}$$

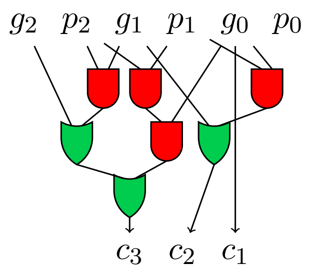
Thus, each carry bit c_{i+1} can be computed by a path-like formula on inputs $g_i, p_i, g_{i-1}, p_{i-1}, \dots, g_1, p_1, g_0$, where gates alternate between AND and OR. We call such functions AND-OR paths (see Definition 2.5.1), and optimizing AND-OR paths is a crucial step in adder optimization and of this thesis.

The circuit that directly emerges from Equation (2.18) applied for computing all carry bits is called a **ripple-carry adder**. A ripple-carry adder on $n = 3$ input pairs is shown in Figure 2.6(a).

Observation 2.4.6. The ripple carry adder for n input pairs has depth and size $2n - 2$.



(a) A ripple-carry adder, see Observation 2.4.6.



(b) An adder circuit arising from applying Remark 2.4.7 to depth-optimum And-Or path circuits.

Figure 2.6: Two adder circuits on $n = 3$ input pairs.

Instead of computing each carry bit by a standard And-Or path circuit as in the ripple carry adder, one might use fast And-Or path circuits to compute each carry bit separately. Figure 2.6(b) gives an example with $n = 3$ input pairs where each carry bit is realized by a depth-optimum And-Or path circuit.

Remark 2.4.7. Given n And-Or path circuits $(AOP_i)_{i=1,\ldots,n}$, where AOP_i is an And-Or path circuit on i input pairs, we can construct an adder circuit on n input pairs. The delay of the arising circuit is $d(AOP_n)$, while its size is $\sum_{i=1}^{n} s(AOP_i)$.

Hence, when the only objective function is delay, the fastest adder circuits (as defined in Definition 2.4.4) can be obtained by applying the best possible And-Or path optimization algorithms to compute all the carry bits separately. If all carry-bits are computed independently, this leads to a size at least quadratic in n as the And-Or paths have at least a size linear in n. Thus, the main task in adder optimization is to find fast adder circuits that still have a linear size. This problem is examined in Chapter 8.

2.5 And-Or Path Circuits

An And-Or path is a function of the form

$$t_0 \vee \Bigg(t_1 \wedge \bigg(t_2 \vee \Big(t_3 \wedge \big(t_4 \vee (t_5 \wedge \ldots)\big)\Big)\bigg)\Bigg) \tag{2.19}$$

or

$$t_0 \wedge \Bigg(t_1 \vee \bigg(t_2 \wedge \Big(t_3 \vee \big(t_4 \wedge (t_5 \vee \ldots)\big)\Big)\bigg)\Bigg) \tag{2.20}$$

for input variables $t = (t_0, \ldots, t_{m-1})$. Note that here, we reverse the indexing compared to the notation used for adder circuits (see Equation (2.18)). We formalize this concept as follows.

Definition 2.5.1. Let Boolean input variables $t = (t_0, \dots, t_{m-1})$ for some $m \in \mathbb{N}$ with $m > 0$ be given. Define the Boolean function $g(t)$ by

$$g(t) = \begin{cases} t_0 & m = 1\,, \\ t_0 \wedge t_1 & m = 2\,, \\ t_0 \wedge \Big(t_1 \vee g\big((t_2, \dots, t_{m-1})\big)\Big) & m \geq 3\,, \end{cases}$$

and let $g^*(t)$ be the dual Boolean function of $g(t)$. We call $g(t)$ and $g^*(t)$ **AND-OR paths** on m inputs. We also call m the **length** of the AND-OR paths $g(t)$ and $g^*(t)$ on input variables $t = (t_0, \dots, t_{m-1})$.

Using Theorem 2.1.31, for $m \in \mathbb{N}_{>0}$, we obtain

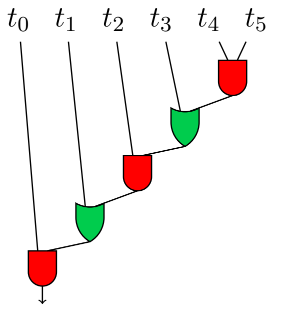
$$g^*(t) = \begin{cases} t_0 & m = 1\,, \\ t_0 \vee t_1 & m = 2\,, \\ t_0 \vee \Big(t_1 \wedge g^*\big((t_2, \dots, t_{m-1})\big)\Big) & m \geq 3\,. \end{cases}$$

Moreover, for $m \in \mathbb{N}_{>1}$, we have

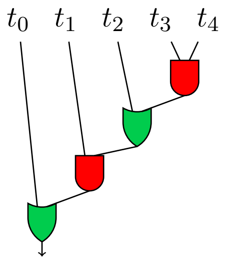
$$g(t) = t_0 \wedge g^*\big((t_1, \dots, t_{m-1})\big)$$

and

$$g^*(t) = t_0 \vee g\big((t_1, \dots, t_{m-1})\big)\,.$$



(a) Standard AND-OR path circuit for $g\big((t_0, \dots, t_5)\big)$.



(b) Standard AND-OR path circuit for $g^*\big((t_0, \dots, t_4)\big)$.

Figure 2.7: Two standard AND-OR path circuits.

In cases when it is irrelevant whether an AND-OR path on inputs t ends with an AND gate or with an OR gate, we often denote the AND-OR path by $h(t)$.

Definition 2.5.2. A circuit realizing an AND-OR path function is called an **AND-OR path circuit**. Given an AND-OR path $h(t)$, we call the Boolean formula for $h(t)$ given by Equation (2.19) or Equation (2.20) a **standard AND-OR path realization** and the corresponding circuit the **standard AND-OR path circuit** for $h(t)$.

The standard circuits for the AND-OR paths $g\big((t_0, \dots, t_5)\big)$ and $g^*\big((t_0, \dots, t_4)\big)$ can be seen in Figure 2.7.

As $g(t)$ and $g^*(t)$ are dual Boolean functions, Theorem 2.1.31 implies the following corollary.

Corollary 2.5.3. *Given Boolean input variables t, any Boolean formula (or circuit) over $\Omega_{\text{mon}} = \{\text{AND2}, \text{OR2}\}$ for $g(t)$ can be transformed into a Boolean formula (or circuit) over $\Omega_{\text{mon}} = \{\text{AND2}, \text{OR2}\}$ for $g^*(t)$ with same delay and size by exchanging all* AND *and* OR *gates and vice versa.* □

We can now formulate one of the main problems considered in this thesis.

AND-OR PATH CIRCUIT OPTIMIZATION PROBLEM

Instance: $m \in \mathbb{N}$, Boolean input variables $t = (t_0, \dots, t_{m-1})$, arrival times $a(t_0), \dots, a(t_{m-1}) \in \mathbb{N}$.

Task: Compute a circuit over $\Omega_{\text{mon}} = \{\text{AND2}, \text{OR2}\}$ realizing $g(t)$ or $g^*(t)$ with minimum possible delay.

By Corollary 2.5.3, this problem is well-defined. Note that finding a depth-optimum AND-OR path circuit is the special case of the AND-OR PATH CIRCUIT OPTIMIZATION PROBLEM when all arrival times are 0, i.e., $a \equiv 0$.

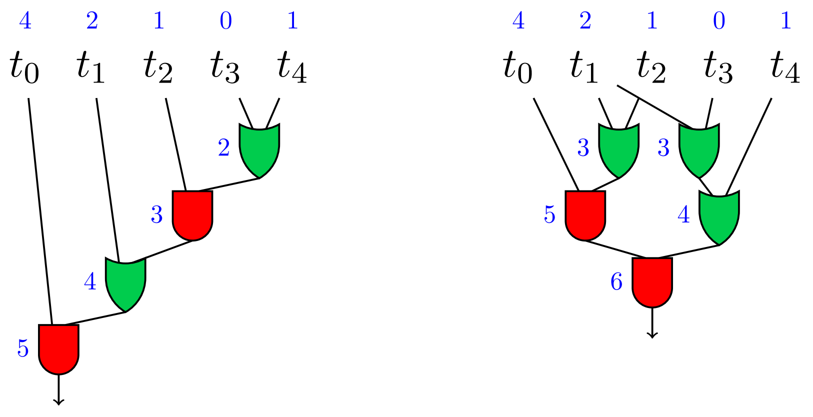
When restricting the set of solutions to formula circuits (or, equivalently, Boolean formulae), we obtain the following problem.

AND-OR PATH FORMULA OPTIMIZATION PROBLEM

Instance: $m \in \mathbb{N}$, Boolean input variables $t = (t_0, \dots, t_{m-1})$, arrival times $a(t_0), \dots, a(t_{m-1}) \in \mathbb{N}$.

Task: A monotone Boolean formula realizing $g(t)$ or $g^*(t)$ with minimum possible delay.

By Theorem 2.3.11, the delays of optimum solutions of the AND-OR PATH CIRCUIT OPTIMIZATION PROBLEM and the AND-OR PATH FORMULA OPTIMIZATION PROBLEM coincide; but note that general circuits might have fewer gates and lower maximum fanouts than formula circuits.



(a) Circuit C_1 with depth$(C_1) = 4$ and delay$(C_1) = 5$.

(b) Circuit C_2 with depth$(C_2) = 3$ and delay$(C_2) = 6$.

Figure 2.8: Two circuits with given input arrival times realizing $g\big((t_0, \dots, t_4)\big)$.

Example 2.5.4. The circuits C_1 and C_2 shown in Figure 2.8(a) and Figure 2.8(b) both realize the AND-OR path $g\big((t_0, \dots, t_4)\big)$. This can be verified by comparing the Boolean formulae corresponding to C_1 and C_2:

$$\begin{aligned}
\phi_{C_1} &= t_0 \wedge \Big(t_1 \vee \big(t_2 \wedge (t_3 \vee t_4)\big)\Big) \\
&\overset{(2.5)}{=} t_0 \wedge \Big(\big(t_1 \vee t_2\big) \wedge \big(t_1 \vee (t_3 \vee t_4)\big)\Big) \\
&\overset{(2.2)}{=} \big(t_0 \wedge (t_1 \vee t_2)\big) \wedge \big((t_1 \vee t_3) \vee t_4\big) \\
&= \phi_{C_2}
\end{aligned}$$

Regarding depth optimization, C_2 is better than C_1, but with respect to the indicated blue arrival times, the delay of C_1 is better than the delay of C_2. For these two concrete instances of the AND-OR PATH FORMULA OPTIMIZATION PROBLEM, we can show that C_1 and C_2 are optimum solutions, respectively: As any binary circuit on 5 inputs has a depth of at least $\lceil \log_2 5 \rceil = 3$, the circuit C_2 is depth-optimum. The delay 5 of C_1 is optimum for the blue arrival times since the input t_0 with arrival time 4 has depth at least 1 in any AND-OR path circuit.

When computing delay-optimum AND-OR path circuits for a given set of arrival times in Chapter 5, the following natural generalization of AND-OR paths arises.

Definition 2.5.5. Let Boolean input variables $t = (t_0, \dots, t_{m-1})$ and an $(m-1)$-tuple $\Gamma = (\circ_0, \dots, \circ_{m-2})$ of gate types $\circ_0, \dots, \circ_{m-2} \in \Omega_{\text{mon}} = \{\text{AND2}, \text{OR2}\}$ be given. We call a Boolean function of the form

$$h(t;\Gamma) := t_0 \circ_0 \Bigg(t_1 \circ_1 \Bigg(t_2 \circ_2 \big(\dots \circ_{m-3} \big(t_{m-2} \circ_{m-2} t_{m-1}\big)\big)\Bigg)\Bigg) \tag{2.21}$$

a **generalized AND-OR path**. Similarly as in Definition 2.5.2, we call the Boolean formula in Equation (2.21) the **standard realization** and the corresponding circuit the **standard circuit** for $h(t;\Gamma)$.

GENERALIZED AND-OR PATH CIRCUIT OPTIMIZATION PROBLEM

Instance: $m \in \mathbb{N}$, Boolean input variables $t = (t_0, \dots, t_{m-1})$, gate types $\Gamma = (\circ_0, \dots, \circ_{m-2})$, arrival times $a(t_0), \dots, a(t_{m-1}) \in \mathbb{R}$.

Task: Compute a circuit over $\Omega_{\text{mon}} = \{\text{AND2}, \text{OR2}\}$ realizing $h(t;\Gamma)$ with minimum possible delay.

Though, our main interest lies in the optimization of AND-OR paths.

In order to understand (generalized) AND-OR paths more thoroughly, it is helpful to divide the inputs into two groups, similarly as in Equation (2.18).

Definition 2.5.6. Let $h(t;\Gamma)$ be a generalized AND-OR path of length $m \geq 1$. An input t_i of $h(t;\Gamma)$ is called a **generate signal** (or a **propagate signal**) if the unique successor of t_i in the standard AND-OR path circuit for $h(t;\Gamma)$ is an OR gate (or an AND gate). The **signal partition** of $h(t;\Gamma)$ is the unique partition $(t_0, \dots, t_{m-1}) = P_0 +\!\!+ \dots +\!\!+ P_c$ of the inputs into maximal consecutive sub-tuples $P_0, \dots, P_c$ called **input groups** such that for each $b \in \{0, \dots, c\}$, the sub-tuple P_b contains only propagate or only generate signals.

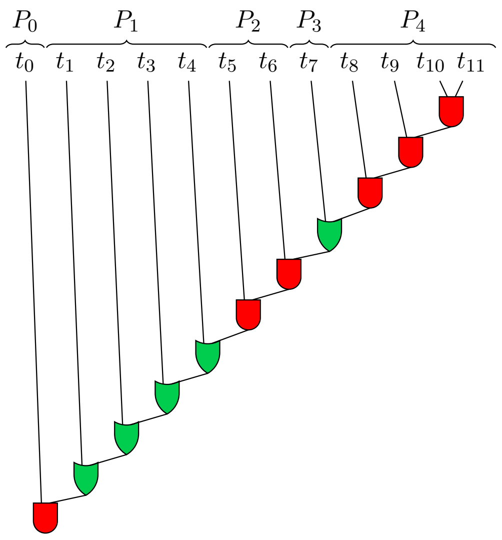

P_0
P_1
P_2
P_3
P_4
t_0 t_1 t_2 t_3 t_4 t_5 t_6 t_7 t_8 t_9 t_{10} t_{11}

Figure 2.9: A generalized AND-OR path with its signal partition.

Figure 2.9 visualizes the signal partition of the inputs for a concrete generalized AND-OR path.

We can now characterize the true points of generalized AND-OR paths.

Proposition 2.5.7. *Let $h(t;\Gamma)$ be a generalized* AND-OR *path of length $m \geq 2$. A value $\alpha \in \{0,1\}^m$ is a true point of $h(t;\Gamma)$ if and only if at least one of the following conditions is fulfilled:*

(i) There is a true generate signal α_i, $i \in \{0,\dots,m-1\}$, and all propagate signals α_j with $j < i$ are true.

(ii) The input t_{m-1} is a propagate signal and all propagate signals in α are true.

Proof. We prove the statement by induction on m.

For $m = 2$, the statement can be verified directly both for $h(t;(\text{AND})) = g(t) = t_0 \wedge t_1$ and $h(t;(\text{OR})) = g^*(t) = t_0 \vee t_1$.

Now consider any $m \geq 3$ and assume that the statement is already proven for all generalized AND-OR paths of length strictly less than m.

Using Definition 2.5.5, we can write $h(t;\Gamma) = t_0 \circ_0 h(t';\Gamma')$, where $t' = (t_1,\dots,t_{m-1})$ and $\Gamma' = (\circ_1,\dots,\circ_{m-2})$.

First assume that $\circ_0 = $ AND. Thus, the true points of $h(t;\Gamma)$ are exactly those $\alpha \in \{0,1\}^m$ with $\alpha_0 = 1$ and $(\alpha_1,\dots,\alpha_{m-1}) \in \big(h(t';\Gamma')\big)^{-1}(1)$. Since t_0 is a propagate signal of $h(t;\Gamma)$, the statement follows by induction hypothesis.

Now, we may assume that $\circ_0 = $ OR. A value $\alpha \in \{0,1\}^m$ is a true point of $h(t;\Gamma)$ if and only if t_0 is true – in which case condition (i) is fulfilled – or if $(\alpha_1,\dots,\alpha_{m-1})$ is a true point of $h(t';\Gamma')$ – in which case the statement is true by induction hypothesis. □

From Proposition 2.5.7, we can deduce the prime implicants of AND-OR paths:

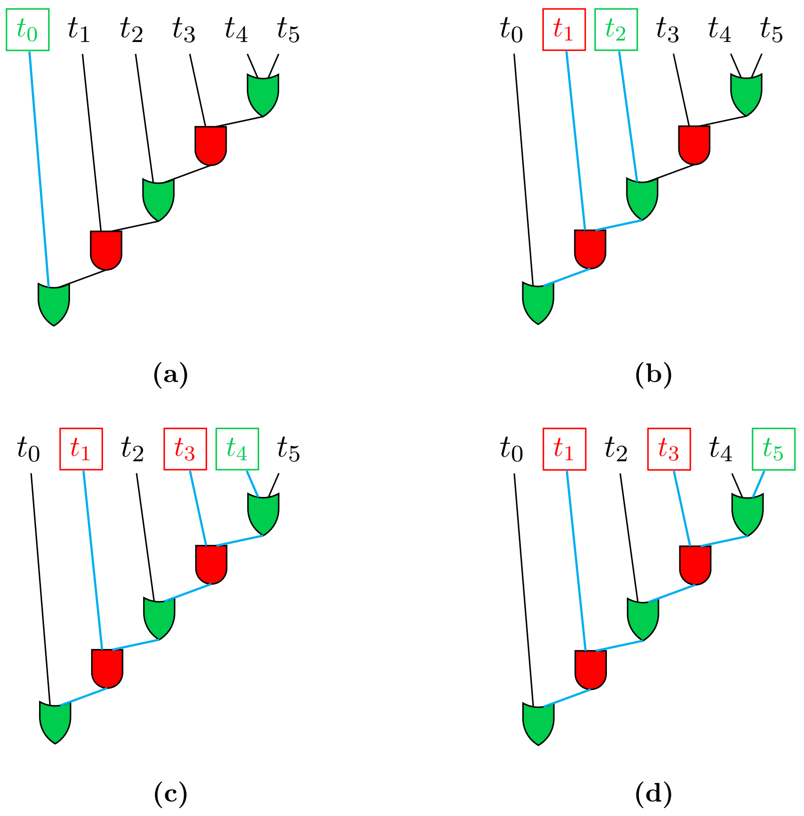


Figure 2.10: All prime implicants of the AND-OR path $f^*\big((t_0, \ldots, t_5)\big)$. Figures 2.10(a) to 2.10(d) illustrate one minimal true point each: The corresponding true inputs are boxed; in green if they are generate signals and in red if they are propagate signals.

Corollary 2.5.8. *Consider a generalized* AND-OR *path $h(t; \Gamma)$ on Boolean input variables $t = (t_0, \ldots, t_{m-1})$. The prime implicants of $h(t; \Gamma)$ are*

$$\left\{ t_i \wedge \bigwedge_{j<i, t_j \text{ propagate signal}} t_j : t_i \text{ generate signal or } i = m-1 \right\}. \qquad \square$$

Figure 2.10 shows all prime implicants for the AND-OR path $f^*\big((t_0, \ldots, t_5)\big)$. An important implication of this corollary is the following statement.

Corollary 2.5.9. *Any generalized* AND-OR *path $h(t; \Gamma)$ on Boolean input variables $t = (t_0, \ldots, t_{m-1})$ depends essentially on all of its inputs.*

Proof. Let t_i with $i \in \{0, \ldots, m-1\}$ be an input of $h(t; \Gamma)$. By Corollary 2.5.8, there is a prime implicant of $h(t; \Gamma)$ that contains t_i. Hence, by Observation 2.1.20, $h(t; \Gamma)$ depends essentially on t_i. $\square$

2.6 Previous Work

When circuit size is not taken into account, delay optimization for AND-OR paths and adders is equally hard: every AND-OR path circuit yields an adder circuit with the same delay and vice versa (cf. Equation (2.18) and Remark 2.4.7). Also, lower bounds on delay hold for both problems simultaneously. Naturally, the adder optimization problem has been studied more widely in previous work.

Section 2.6.1 summarizes the previously known lower bounds on depth and delay for AND-OR path circuits and adder circuits. In Section 2.6.2, we present well-known strategies to construct AND-OR path circuits in a recursive fashion, i.e., to construct circuits for AND-OR paths of length m based on circuits for AND-OR paths with length strictly smaller than m. These strategies are the basic ingredient for all AND-OR path and adder optimization algorithms. In Sections 2.6.3 and 2.6.4, we will discuss the previously best known algorithms for depth and delay optimization of AND-OR paths and adders, respectively.

2.6.1 Lower Bounds

A simple lower bound on the delay of any solution to the AND-OR PATH CIRCUIT OPTIMIZATION PROBLEM and the AND-OR PATH FORMULA OPTIMIZATION PROBLEM can be derived from Theorem 2.3.15 as follows.

Proposition 2.6.1. *The delay of any* AND-OR *path circuit over* $\Omega_{\text{mon}} = \{\text{AND2}, \text{OR2}\}$ *on inputs* $t = (t_0, \dots, t_{m-1})$ *with arrival times* $a(t_0), \dots, a(t_{m-1}) \in \mathbb{N}$ *is at least* $\lceil \log_2 W \rceil$*, where* $W := \sum_{i=0}^{m-1} 2^{a(t_i)}$ *as defined in Definition 2.3.16.*

Proof. Since AND-OR paths depend essentially on all their inputs by Corollary 2.5.9, the lower bound $\lceil \log_2 W \rceil$ on the delay of any binary circuit shown in Theorem 2.3.15 is valid for AND-OR path circuits. □

For arbitrary arrival times, Proposition 2.6.1 gives the best known lower bound on the delay of AND-OR path circuits. But regarding the asymptotic behavior of the depth of AND-OR path circuits, there are several interesting results. Commentz-Walter, partially together with Sattler, showed lower bounds on the product of size and depth of any monotone [Com79] and non-monotone [CS80] Boolean formula. In these two works, only AND-OR paths ending with an AND gate are considered, and they have $m = 2n$ inputs. Hence, we use this notation in the this section.

Definition 2.6.2. Given $m \in \mathbb{N}$, the **product complexity** of AND-OR paths is

$$P_{\text{mon}}(n) := \min\Big\{ \text{size}(\phi) \cdot \text{depth}(\phi) : \phi \text{ is a Boolean formula for } g\big((t_0, \dots, t_{2n-1})\big) \Big\},$$

and the **monotone product complexity** is

$$P_{\text{nmon}}(n) := \min\Big\{ \text{size}(\phi) \cdot \text{depth}(\phi) : \phi \text{ is a monotone Boolean formula for } g\big((t_0, \dots, t_{2n-1})\big) \Big\}.$$

Theorem 2.6.3 (Commentz-Walter and Sattler [CS80])**.** *For any $n \in \mathbb{N}$ with $n \geq 17$, we have*

$$P_{\text{nmon}}(n) \geq \frac{n \log_2 n \log_2 \log_2 n}{8 \log_2 \log_2 \log_2 \log_2 n}.$$

From this, a lower bound on the depth of AND-OR path circuits can be deduced easily as follows.

Corollary 2.6.4. *For any $\alpha \in (0,1)$, there is $N_\alpha \in \mathbb{N}$ such that for all $n \in \mathbb{N}$, $n \geq N_\alpha$, and any circuit C over $\Omega_{\text{nmon}} = \{\textsc{And2}, \textsc{Or2}, \textsc{Inv}\}$ for an* And-Or *path of length $2n$, we have*

$$\operatorname{depth}(C) \geq \log_2 n + \alpha \log_2 \log_2 \log_2 n\,.$$

Proof. Consider $\alpha \in (0,1)$ and assume that for all $N_\alpha \in \mathbb{N}$, there is $n \geq N_\alpha$ such that an And-Or path circuit C over Ω_{nmon} for $g\big((t_0, \dots, t_{2n-1})\big)$ with $\operatorname{depth}(C) < \log_2 n + \alpha \log_2 \log_2 \log_2 n$ exists. Choose N_α such that for every $n \geq N_\alpha$, we have $(\log_2 \log_2 n)^{1-\alpha} \geq 16 \log_2 \log_2 \log_2 \log_2 n$. There is some $n \geq N_\alpha$ and a circuit C for $g\big((t_0, \dots, t_{2n-1})\big)$ with

$$\operatorname{depth}(C) < \log_2 n + \alpha \log_2 \log_2 \log_2 n\,. \tag{2.22}$$

By Corollary 2.3.12, the Boolean formula ϕ corresponding to C fulfills $\operatorname{depth}(\phi) = \operatorname{depth}(C)$. Hence, we have

$$\begin{aligned}
\mathrm{P}_{\text{nmon}}(n) &\leq \operatorname{size}(\phi) \cdot \operatorname{depth}(\phi) \\
&\overset{\text{Obs. 2.3.5}}{\leq} 2^{\operatorname{depth}(\phi)} \operatorname{depth}(\phi) \\
&\overset{\substack{\operatorname{depth}(\phi) = \operatorname{depth}(C),\\ (2.22)}}{<} 2^{\log_2 n + \alpha \log_2 \log_2 \log_2 n} (\log_2 n + \alpha \log_2 \log_2 \log_2 n) \\
&= n (\log_2 \log_2 n)^\alpha (\log_2 n + \alpha \log_2 \log_2 \log_2 n) \\
&\overset{\alpha \leq 1}{\leq} 2n \log_2 n (\log_2 \log_2 n)^\alpha\,.
\end{aligned}$$

Together with the lower bound on $\mathrm{P}_{\text{nmon}}(n)$ given by Theorem 2.6.3, this yields

$$(\log_2 \log_2 n)^{1-\alpha} < 16 \log_2 \log_2 \log_2 \log_2 n\,,$$

a contradiction to the choice of N_α. □

For $\alpha = 0.15$, Khrapchenko [Khr07] makes this statement more precise.

Corollary 2.6.5 (Khrapchenko [Khr07])**.** *For any $n \geq 2^{2^{32}}$ and any* And-Or *path circuit C over Ω_{nmon} on $2n$ inputs, we have*

$$\operatorname{depth}(C) \geq \log_2 n + 0.15 \log_2 \log_2 \log_2 n\,.$$

When restricting the set of possible solutions to monotone formulae, Commentz-Walter shows a stronger lower bound and a matching upper bound on the product complexity.

Theorem 2.6.6 (Commentz-Walter [Com79])**.** *We have $\mathrm{P}_{\text{mon}}(n) \in \Theta(n \log_2^2 n)$.*

In this case, she also derives an implied bound on the depth of monotone And-Or path realizations.

Corollary 2.6.7 (Commentz-Walter [Com79])**.** *Let $d(n)$ denote the optimum depth of any circuit over Ω_{mon} realizing $g\big((t_0, \dots, t_{2n-1})\big)$. We have*

$$d(n) = \log_2 n + \Omega(\log_2 \log_2 n)\,.$$

From Theorem 2.6.6, we can also derive the following lower bound.

Corollary 2.6.8. *There is some $\alpha \in \mathbb{R}$ and some $N_\alpha \in \mathbb{N}$ such that for all $n \geq N_\alpha$, any circuit C over Ω_{mon} for $g\big((t_0, \dots, t_{2n-1})\big)$ fulfills*

$$\operatorname{depth}(C) \geq \log_2 n + \log_2 \log_2 n + \alpha\,.$$

Proof. Assume on the contrary that for all $\alpha \in \mathbb{R}$ and for all $N_\alpha \in \mathbb{N}$, there is $n \geq N_\alpha$ and a circuit C for $g\big((t_0, \dots, t_{2n-1})\big)$ with $\operatorname{depth}(C) < \log_2 n + \log_2 \log_2 n + \alpha$.

By Theorem 2.6.6, there are $\beta \in \mathbb{R}_{>0}$ and $N_\beta \in \mathbb{N}$ such that for all $n \geq N_\beta$, we have

$$\mathrm{P}_{\text{nmon}}(n) \geq \beta n \log_2^2 n\,. \tag{2.23}$$

Choose $\alpha := \min\{-1, \log_2(\beta) - 1\} \in \mathbb{R}_{<0}$ and $N_\alpha := N_\beta$. Let $n \geq N_\alpha$ such that a circuit C of $g\big((t_0, \dots, t_{2n-1})\big)$ with

$$\operatorname{depth}(C) < \log_2 n + \log_2 \log_2 n + \alpha\,. \tag{2.24}$$

exists. By Corollary 2.3.12, the Boolean formula ϕ corresponding to C fulfills $\operatorname{depth}(\phi) = \operatorname{depth}(C)$. We conclude

$$\begin{aligned}
\operatorname{size}(\phi) \cdot \operatorname{depth}(\phi) &\overset{\text{Obs. 2.3.5}}{\leq} 2^{\operatorname{depth}(\phi)} \operatorname{depth}(\phi) \\
&\overset{\substack{\operatorname{depth}(\phi)=\operatorname{depth}(C),\\ (2.24)}}{<} n \log_2 n 2^\alpha (\log_2 n + \log_2 \log_2 n + \alpha) \\
&\overset{\alpha<0}{<} 2^{\alpha+1} n \log_2^2 n \\
&\overset{\alpha<\log_2(\beta)-1}{\leq} \beta n \log_2^2 n\,,
\end{aligned}$$

which contradicts (2.23). □

Hitzschke [Hit18] showed that asymptotically, the additive constant α in Corollary 2.6.8 can be chosen arbitrarily close to -4:

Remark 2.6.9. Hitzschke [Hit18] specified and improved the lower bounds on the depth of AND-OR path circuits by Commentz-Walter [Com79]. We review his result for monotone circuits. Assume that n is of the form $n = 2^{2^k}$ with $k \in \mathbb{N}$. Only for $k \geq 7$, Hitzschke's (and thus Commentz-Walter's) lower bound is stronger than the lower bound $\lceil \log_2(2n) \rceil$ on the depth of any binary circuit on $2n$ inputs. For $k \geq 18$, Hitzschke shows that $\log_2 n + \log_2 \log_2 n - 4.01$ is a lower bound on the depth of any AND-OR path circuit over Ω_{mon} on n inputs, and that asymptotically, the additive constant approaches -4.

Hence, apart from Proposition 2.6.1, none of the statements in this section yields a lower bound on the depth of AND-OR path formulae for a small number of inputs. In order to check whether a certain realization has optimum delay, Proposition 2.6.1 remains the only lower bound known.

2.6.2 Recursion Strategies

For AND-OR paths with a very short length, depth-optimum or even delay-optimum formulae or circuits are easy to find.

Observation 2.6.10. In Proposition 5.2.6, we will see that for $m \leq 3$, the standard realization is delay-optimum for any prescribed arrival times $a(t_0), \dots, a(t_{m-1}) \in \mathbb{N}$; and that for $m = 4$, the standard realization of $g(t)$ with depth 3 is depth-optimum.

For $m = 5$, Figure 2.8(b) (page 38) shows a realization for $g(t)$ with depth 3. This is the optimum depth achievable for this instance as by Proposition 2.6.1, a realization with depth 2 does not exist.

For larger m, a common strategy to find And-Or path realizations with good delay is to reduce the problem to the construction of And-Or paths of strictly smaller lengths. In order to derive several approaches that follow this general strategy, an important ingredient is the characterization of the true points (see Definition 2.1.1) of $g(t)$ and $g^*(t)$ from Proposition 2.5.7.

First, we shall see that the characterization of true points immediately yields a circuit for an And-Or path of length m with logarithmic delay.

Proposition 2.6.11. *Consider an* And-Or *path $h(t)$ with length m, and let arrival times $a(t_0), \dots, a(t_{m-1}) \in \mathbb{N}$ be given. There is a circuit C for $h(t)$ with delay*

$$\operatorname{delay}(C) \leq \log_2\big(W(t;a)\big) + \log_2 m + 2\,.$$

Proof. Assume first that t_{m-1} is a generate signal. By Proposition 2.5.7, we have

$$h(t) = \bigvee_{t_i \text{ generate signal}} t_i \wedge \left(\bigwedge_{t_j \text{ propagate signal, } j<i} t_j \right). \tag{2.25}$$

Now, we construct a circuit C realizing $h(t)$: For a fixed generate signal t_i, we can use Theorem 2.3.21 in order to compute a delay-optimum circuit for $t_i \wedge \bigwedge_{t_j \text{ propagate signal, } j<i} t_j$ with delay at most

$$\left\lceil \log_2 \left(2^{t_i} + \sum_{t_j \text{ propagate signal, } j<i} 2^{t_j} \right) \right\rceil \leq \left\lceil \log_2\big(W(t;a)\big) \right\rceil.$$

Based on this, each input of the Or in Equation (2.25) thus has arrival time at most $\left\lceil \log_2\big(W(t;a)\big) \right\rceil$. Applying Theorem 2.3.21 again to construct the Or-tree, the resulting circuit C has a delay of at most

$$\begin{aligned} \left\lceil \log_2 \left(\sum_{t_i \text{ generate signal}} 2^{\left\lceil \log_2(W(t;a)) \right\rceil} \right) \right\rceil &\leq \log_2\Big(m 2^{\log_2(W(t;a))+1}\Big) + 1 \\ &= \log_2\big(2mW(t;a)\big) + 1 \\ &= \log_2\big(W(t;a)\big) + \log_2 m + 2\,. \end{aligned}$$

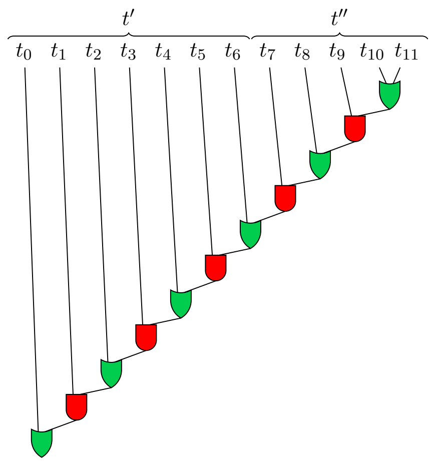
If t_{n-1} is a propagate signal of $h(t)$, then t_{n-1} is a generate signal for the dual function $h^*(t)$. Hence, by the first part, we can find a realization ϕ for $h^*(t)$ with delay at most $\log_2\big(W(t;a)\big) + \log_2 m + 2$, and by Theorem 2.1.31, ϕ^* is a realization of $h(t) \overset{\text{Prop. 2.1.30}}{=} \big(h^*(t)\big)^*$ with the same delay. $\square$

Plugging together Propositions 2.6.1 and 2.6.11 and using that the delay of a circuit with integral arrival times is integral, we obtain the following corollary.

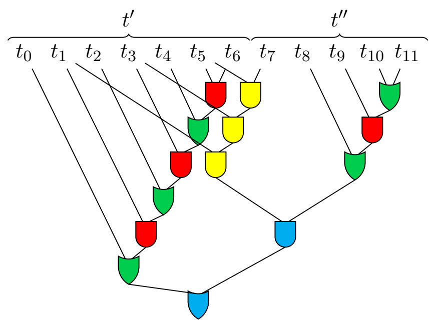
Corollary 2.6.12. *Let an* And-Or *path $h(t)$ on inputs $t = (t_0, \dots, t_{m-1})$ with arrival times $a(t_0), \dots, a(t_{m-1}) \in \mathbb{N}$ and a delay-optimum circuit C realizing $h(t)$ with respect to arrival times a be given. We have*

$$\left\lceil \log_2\big(W(t;a)\big) \right\rceil \leq \operatorname{delay}(C) \leq \left\lfloor \log_2\big(W(t;a)\big) + \log_2 m \right\rfloor + 2\,. \qquad \square$$

In the rest of this section, we shall see recursion formulas that help improving the upper bound on the optimum delay. All these recursion formulas can be found in Grinchuk [Gri08] (although with different proofs), and in different form and for certain special cases also in earlier works. In particular, we will use the characterization of the true points of AND-OR paths given in Proposition 2.5.7 in order to describe several well-known variants of a recursive strategy to optimize AND-OR paths. The key idea for these is depicted in Figure 2.11 and proven in Lemma 2.6.13.


t'
t''
t_0 t_1 t_2 t_3 t_4 t_5 t_6 t_7 t_8 t_9 t_{10} t_{11}

(a) The standard AND-OR path circuit for $g^*\big((t_0, \dots, t_{11})\big)$.


t'
t''
t_0 t_1 t_2 t_3 t_4 t_5 t_6 t_7 t_8 t_9 t_{10} t_{11}

(b) The circuit for $g^*\big((t_0, \dots, t_{11})\big)$ as indicated by Lemma 2.6.13 with $k = 7$.

Figure 2.11: Illustration of the split from Lemma 2.6.13.

Lemma 2.6.13. *Let input variables $t = (t_0, \dots, t_{m-1})$ and an odd integer k with $1 \leq k < m$ be given. Then, we have*

$$g^*(t) = g^*\big((t_0, \dots, t_{k-1})\big) \vee \Big(\operatorname{sym}\big((t_1, t_3, \dots, t_k)\big) \wedge g^*\big((t_{k+1}, \dots, t_{m-1})\big)\Big)$$

and

$$g(t) = g\big((t_0, \dots, t_{k-1})\big) \wedge \Big(\operatorname{sym}^*\big((t_1, t_3, \dots, t_k)\big) \vee g\big((t_{k+1}, \dots, t_{m-1})\big)\Big)\,.$$

Proof. By Corollary 2.5.3, it suffices to prove the first statement. For this, we show that the true points of $g^*(t)$ as given in Proposition 2.5.7 are exactly the true points of the function

$$h(t) := g^*\big((t_0, \dots, t_{k-1})\big) \vee \Big(\operatorname{sym}\big((t_1, t_3, \dots, t_k)\big) \wedge g^*\big((t_{k+1}, \dots, t_{m-1})\big)\Big)\,. \tag{2.26}$$

This is done in the subsequent claims.

Claim 1. Any true point $\alpha \in \{0,1\}^m$ of $g^*(t)$ is a true point of $h(t)$.

Proof of claim: First, we consider only true points α of $g^*(t)$ that fulfill condition (i) of Proposition 2.5.7 with a true generate signal α_i with $i \leq k-1$ for which all propagate signals α_j with $j < i$ are true. Then, $(\alpha_0, \dots, \alpha_{k-1})$ is a true point of $g^*\big((t_0, \dots, t_{k-1})\big)$ by Proposition 2.5.7, and, by definition of $h(t)$, a true point of $h(t)$.

Secondly, consider all other true points α of $g^*(t)$. By Proposition 2.5.7, $\alpha_1, \alpha_3, \dots, \alpha_k$ must all be true, and $(\alpha_k, \dots, \alpha_{m-1})$ must be a true point of $g^*\big((t_{k+1}, \dots, t_{m-1})\big)$. Hence, by Equation (2.26), α is also a true point of $h(t)$. ◻

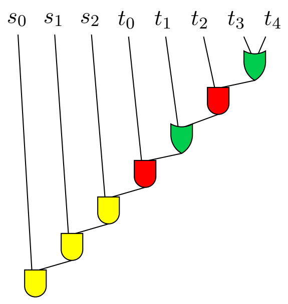


Figure 2.12: The standard circuit of the extended AND-OR path $f\big((s_0, s_1, s_2), (t_0, \dots, t_4)\big)$. We have $n = 3$ and $m = 5$. The gates fed by alternating inputs are colored red (AND) and green (OR); the gates fed by symmetric inputs are colored yellow.

Claim 2. Any true point $\alpha \in \{0, 1\}^n$ of $h(t)$ is a true point of $g^*(t)$.
Proof of claim: In the realization of $h(t)$, the gate preceding the final output is an OR gate. Thus, if $\alpha \in \{0, 1\}^n$ is a true point of $h(t)$, then one of the two sub-circuits of this OR gate must have a true output.

If $g^*\Big((\alpha_0, \dots, \alpha_{k-1})\Big) = 1$, then $(\alpha_0, \dots, \alpha_{k-1})$ is a true point of the function $g^*\Big((\alpha_0, \dots, \alpha_{k-1})\Big)$ and thus α a true point of $g^*((\alpha_0, \dots, \alpha_{m-1}))$ by Proposition 2.5.7.

On the other hand, if $\Big(\mathrm{sym}\big((\alpha_1, \alpha_3, \dots, \alpha_k)\big) \wedge g^*\big((\alpha_{k+1}, \dots, \alpha_{m-1})\big)\Big) = 1$, then $\alpha_1, \alpha_3, \dots, \alpha_k$ are all true and $g^*\big((\alpha_{k+1}, \dots, \alpha_{m-1})\big)$ is true. Hence, by Proposition 2.5.7, α is a true point of $f^*(t)$. □

□

In particular, Lemma 2.6.13 together with Remark 2.1.14 implies that once we have found realizations ϕ of $g^*\big((t_0, \dots, t_{k-1})\big)$, ψ of $\mathrm{sym}\big((t_1, t_3, \dots, t_k)\big)$ and τ of $g^*\big((t_{k+1}, \dots, t_{m-1})\big)$, the Boolean formula $\phi \vee (\psi \wedge \tau)$ realizes $g^*(t)$. One way to do this would be to compute ϕ and τ by recursively applying Lemma 2.6.13 and by computing ψ using Huffman coding as described in Theorem 2.3.21.

However, it turns out to be beneficial to consider the function

$$\mathrm{sym}\big((t_1, t_3, \dots, t_k)\big) \wedge g^*\big((t_{k+1}, \dots, t_{m-1})\big)$$

as one entity instead of searching for realizations for $\mathrm{sym}\big((t_1, t_3, \dots, t_k)\big)$ and $g^*\big((t_{k+1}, \dots, t_{m-1})\big)$ separately. Thus, we introduce the following definition.

Definition 2.6.14. Let $n, m \in \mathbb{N}$ with $m > 0$ and Boolean input variables $s = (s_0, \dots, s_{n-1})$ and $t = (t_0, \dots, t_{m-1})$ be given. We call each of the functions

$$f(s, t) = \mathrm{sym}(s) \wedge g(t) \quad \text{and} \quad f^*(s, t) = \mathrm{sym}^*(s) \vee g^*(t)$$

an **extended AND-OR path** on $n + m$ inputs. We call t the **alternating inputs** and s the **symmetric inputs** of the extended AND-OR paths $f(s, t)$ and $f^*(s, t)$.

Note that each extended AND-OR path is a generalized AND-OR path, see Definition 2.5.5. In particular, they also have standard realizations and standard circuits, and they depend essentially on all of their inputs (see Definition 2.5.5

and Corollary 2.5.9). Figure 2.12 shows the standard circuit for an extended AND-OR path with 3 symmetric inputs and 5 alternating inputs.

In order to reformulate Lemma 2.6.13 using extended AND-OR paths in a compact way, we define a subset of input variables that contains each second input variable.

Definition 2.6.15. Given input variables $t = (t_0, \dots, t_{m-1})$, we define the input variables

$$\widehat{t} := \begin{cases} (t_1, t_3, t_5, \dots, t_{m-2}) & \text{for } m \text{ odd,} \\ (t_0, t_2, t_4, \dots, t_{m-2}) & \text{for } m \text{ even.} \end{cases}$$

Now, we can generalize Lemma 2.6.13 as follows.

Corollary 2.6.16. *Let input variables $t = (t_0, \dots, t_{m-1})$ and an odd integer k with $1 \leq k < m$ be given. Denote by t' the odd-length prefix $t' = (t_0, t_1, \dots, t_{k-1})$ of t, and by t'' the remaining inputs of t, i.e., $t'' = (t_k, \dots, t_{m-1})$. Then, we have*

$$g^*(t) = g^*\big(t'\big) \lor f\Big(\widehat{t'}, t''\Big) \quad \text{and} \quad g(t) = g\big(t'\big) \land f^*\Big(\widehat{t'}, t''\Big) .$$

Proof. By Corollary 2.5.3, it suffices to prove the first statement. We have

$$\begin{aligned} g^*(t) &\overset{\text{Lem. 2.6.13}}{=} g^*\big((t_0, \dots, t_{k-1})\big) \lor \Big(\text{sym}\big((t_1, t_3, \dots t_k)\big) \land g^*\big((t_{k+1}, \dots, t_{m-1})\big)\Big) \\ &= g^*\big(t'\big) \lor \bigg(\text{sym}\big((t_1, t_3, \dots t_{k-2})\big) \land \Big(t_k \land g^*\big((t_{k+1}, \dots, t_{m-1})\big)\Big)\bigg) \\ &\overset{\text{Def. 2.5.1}}{=} g^*\big(t'\big) \lor \bigg(\text{sym}\Big(\widehat{t'}\Big) \land g\big((t_k, \dots, t_{m-1})\big)\bigg) \\ &\overset{\substack{\text{Def. 2.6.15,} \\ \text{Def. 2.6.14}}}{=} g^*\big(t'\big) \lor f\Big(\widehat{t'}, t''\Big) . \end{aligned}$$ □

Note that this is a generalization of Lemma 2.6.13 because the formula $\text{sym}\big((t_1, t_3, \dots, t_k)\big) \land f^*(t_{k+1}, \dots, t_{m-1})$ restricts the set of possible realizations of the function $f\Big(\widehat{t'}, t''\Big)$ to those that arise from realizations for $\text{sym}\big((t_1, t_3, \dots, t_k)\big)$ and $f^*(t_{k+1}, \dots, t_{m-1})$ concatenated by an AND. Now, we have the freedom to realize $f\Big(\widehat{t'}, t''\Big)$ arbitrarily.

Since the realizations for AND-OR paths implied by Corollary 2.6.16 are based on realizations for extended AND-OR paths, we generalize this statement such that it also can compute realizations for extended AND-OR paths. We will call the arising method to realize extended AND-OR paths in a recursive fashion an **alternating split** because it leaves the symmetric inputs of the original extended AND-OR path untouched while the alternating inputs are split into two groups.

Corollary 2.6.17 (Alternating split, odd prefix)**.** *Let Boolean input variables $s = (s_0, \dots, s_{n-1})$ and $t = (t_0, \dots, t_{m-1})$ and an odd integer k with $1 \leq k < m$ be given. Denote by t' the odd-length prefix $t' = (t_0, t_1, \dots, t_{k-1})$ of t, and by t'' the remaining inputs of t, i.e., $t'' = (t_k, \dots, t_{m-1})$. Then, we have*

$$f^*(s, t) = f^*\big(s, t'\big) \lor f\Big(\widehat{t'}, t''\Big) \tag{2.27}$$

and

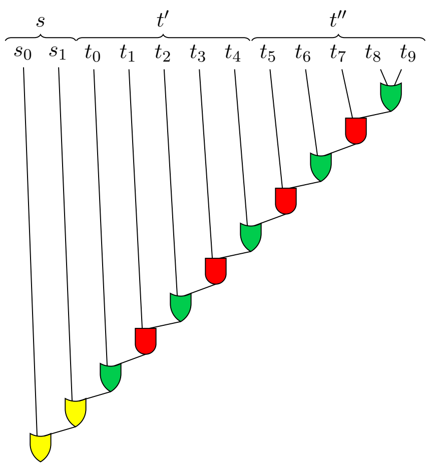
$$f(s, t) = f\big(s, t'\big) \land f^*\Big(\widehat{t'}, t''\Big) . \tag{2.28}$$

Proof. Due to Corollary 2.5.3, it suffices to prove the first statement. Using Corollary 2.6.16 and Definition 2.6.14, this holds due to

$$\begin{aligned} f^*(s,t) &\overset{\text{Def. 2.6.14}}{=} \operatorname{sym}^*(s) \vee g^*(t) \\ &\overset{\text{Cor. 2.6.16}}{=} \operatorname{sym}^*(s) \vee \left(g^*(t') \vee f\left(\widehat{t'}, t''\right)\right) \\ &\overset{(2.2)}{=} \left(\operatorname{sym}^*(s) \vee g^*(t')\right) \vee f\left(\widehat{t'}, t''\right) \\ &\overset{\text{Def. 2.6.14}}{=} f^*(s,t') \vee f\left(\widehat{t'}, t''\right). \end{aligned}$$

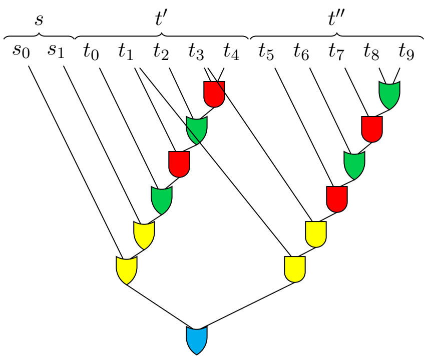
□

Figure 2.13 shows an illustration of the alternating split with a prefix of length 5. We use standard circuits for $f^*\big((s_0, s_1)(t_0, \dots, t_4)\big)$ and $f^*\big((s_1, t_1, t_3), (t_5, \dots, t_9)\big)$ for illustration purposes. Any circuits realizing these functions could be used here. Recall that one possible realization for the latter function appears in Figure 2.11(b).



s
t'
t''
s_0 s_1 t_0 t_1 t_2 t_3 t_4 t_5 t_6 t_7 t_8 t_9

(a) The standard And-Or path circuit for the extended And-Or path $f^*\big((s_0, s_1), (t_0, \dots, t_9)\big)$.



s
t'
t''
s_0 s_1 t_0 t_1 t_2 t_3 t_4 t_5 t_6 t_7 t_8 t_9

(b) A circuit realizing the function $f^*\big((s_0, s_1), (t_0, \dots, t_9)\big)$ as indicated by Corollary 2.6.17 with $k = 5$.

Figure 2.13: Illustration of the alternating split with an odd prefix.

There is a slightly different split that allows splitting off an even-length prefix of t. Here, we use the notation $x + y := \big(x_0, \dots, x_{q-1}, y_0, \dots, y_{r-1}\big)$ to concatenate two tuples of disjoint input variables $x = \big(x_0, \dots, x_{q-1}\big)$ and $y = (y_0, \dots, y_{r-1})$; and for input variables x, x' with $x = \big(x_0, \dots, x_{q-1}\big)$ and $x' = \big(x_0, \dots, x_{q-1}\big)$ with $q \le r$, we write $x \setminus x' := \big(x_q, \dots, x_{r-q}\big)$.

Corollary 2.6.18 (Alternating split, even prefix)**.** *Let Boolean input variables $s = (s_0, \dots, s_{n-1})$ and $t = (t_0, \dots, t_{m-1})$ and an even integer k with $2 \le k < m$ be given. Denote by t' the even-length prefix $t' = (t_0, t_1, \dots, t_{k-1})$ of t, and by t'' the remaining inputs of t, i.e., $t'' = t \setminus t'$. Then, we have*

$$f^*(s,t) = f^*(s,t') \wedge f^*\left(s + \widehat{t'}, t''\right) \tag{2.29}$$

and

$$f(s,t) = f(s,t') \vee f\left(s + \widehat{t'}, t''\right). \tag{2.30}$$

Proof. Again, it suffices to prove the first statement. We first consider the case that $s = ()$. Here, we apply the alternating split with an odd prefix of length $k+1$ (Corollary 2.6.17) to the modified instance arising from t by adding an auxiliary input variable t_{-1}. This yields the realization

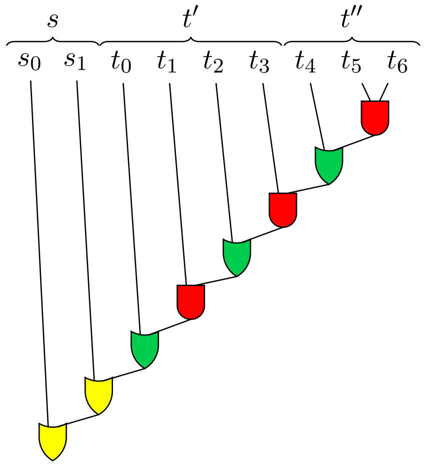
$$\begin{aligned} &g\big((t_{-1}, t_0, \dots, t_{m-1})\big) = \\ &\qquad g\big((t_{-1}, \dots, t_{k-1})\big) \wedge f^*\big((t_0, t_2, t_4, \dots, t_{k-2}), (t_k, \dots, t_{m-1})\big) . \end{aligned} \tag{2.31}$$

Form this, we deduce a realization for $g^*(t)$ since

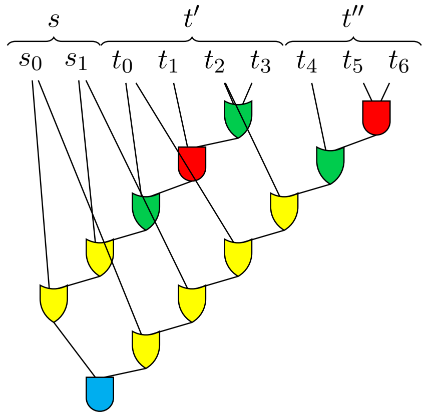
$$\begin{aligned} g^*(t) &\overset{\text{Def. 2.5.1}}{=} g\,|_{t_{-1}=1}\,\big((t_{-1}, t_0, \dots, t_{m-1})\big) \\ &\overset{(2.31)}{=} g\,|_{t_{-1}=1}\,\big((t_{-1}, t_0, \dots, t_{k-1})\big) \wedge f^*\big((t_0, t_2, t_4, \dots, t_{k-2}), (t_k, \dots, t_{m-1})\big) \\ &\overset{\substack{\text{Def. 2.5.1,}\\ \text{Def. 2.6.15}}}{=} g^*(t') \wedge f^*\left(\widehat{t'}, t''\right) . \end{aligned} \tag{2.32}$$

This proves Equation (2.29) in the case that $s = ()$. For arbitrary s, we have

$$\begin{aligned} f^*(s,t) &\overset{\text{Def. 2.6.14}}{=} \text{sym}^*(s) \vee g^*(t) \\ &\overset{(2.32)}{=} \text{sym}^*(s) \vee \left(g^*(t') \wedge f^*\left(\widehat{t'}, t''\right)\right) \\ &\overset{(2.5)}{=} \left(\text{sym}^*(s) \vee g^*(t')\right) \wedge \left(\text{sym}^*(s) \vee f^*\left(\widehat{t'}, t''\right)\right) \\ &\overset{\text{Def. 2.6.14}}{=} f^*(s, t') \wedge f^*\left(s \mathbin{+\!\!+} \widehat{t'}, t''\right) . \end{aligned}$$ □



(a) The standard AND-OR path circuit for $f^*\big((s_0, s_1), (t_0, \dots, t_6)\big)$.



(b) A circuit for $f^*\big((s_0, s_1), (t_0, \dots, t_6)\big)$ as in Corollary 2.6.18 with $k = 4$.

Figure 2.14: Illustration of the alternating split with an even prefix.

Figure 2.14 illustrates the alternating split with an even prefix t' on an extended AND-OR path with 2 symmetric inputs and 7 alternating inputs. Note that for non-trivial symmetric inputs s, the alternating split is much more convenient in the case that the prefix is odd since for an even prefix, the symmetric inputs s appear in both recursive realizations.

Similarly as Lemma 2.6.13 is a special way to use the alternating split from Corollary 2.6.17, the following recursion formula is a special case of the realization in Corollary 2.6.18.

Corollary 2.6.19. *Let Boolean input variables $t = (t_0, \dots, t_{m-1})$ and an even integer k with $2 \leq k < m$ be given. Then, we have*

$$g^*(t) = g^*\big((t_0, t_1, \dots, t_{k-1})\big) \land \Big(\mathrm{sym}^*\big((t_0, t_2, \dots, t_{k-2})\big) \lor g^*\big((t_k, \dots, t_{m-1})\big)\Big)$$

and

$$g(t) = g\big((t_0, t_1, \dots, t_{k-1})\big) \lor \Big(\mathrm{sym}\big((t_0, t_2, \dots, t_{k-2})\big) \land g\big((t_k, \dots, t_{m-1})\big)\Big)\,. \qquad \square$$

The definition of extended AND-OR paths implies other ways to realize extended AND-OR paths recursively. We shall call these **symmetric splits**.

Observation 2.6.20 (Symmetric splits)**.** Given Boolean input variables $s = (s_0, \dots, s_{n-1})$ and $t = (t_0, \dots, t_{m-1})$, we have

$$f(s,t) = \mathrm{sym}(s) \land g(t)\,, \tag{2.33}$$
$$f^*(s,t) = \mathrm{sym}^*(s) \lor g^*(t)\,. \tag{2.34}$$

Furthermore, if $k < n$, we have

$$f(s,t) = \mathrm{sym}\big((s_0, \dots, s_{k-1})\big) \land f\big((s_k, \dots, s_{n-1}), t\big) \tag{2.35}$$

and

$$f^*(s,t) = \mathrm{sym}^*\big((s_0, \dots, s_{k-1})\big) \lor f^*\big((s_k, \dots, s_{n-1}), t\big)\,. \tag{2.36}$$

More generally, for $k \leq n$ and any k-elemental sub-tuple $s' = (s_{i_0}, \dots, s_{i_{k-1}})$ of s, we have

$$f(s,t) = \mathrm{sym}(s') \land f\big(s \backslash s', t\big) \tag{2.37}$$

and

$$f^*(s,t) = \mathrm{sym}^*(s') \land f^*\big(s \backslash s', t\big)\,. \tag{2.38}$$

If m is small, $f(s,t)$ and $f^*(s,t)$ are actually symmetric functions by Definitions 2.5.1 and 2.6.14. Hence, in this case, we can compute delay-optimum solutions for them by Theorem 2.3.21:

Observation 2.6.21. Assume that $m \leq 2$ – hence both $f(s,t)$ and $f^*(s,t)$ are symmetric functions – and that all input arrival times are integral. Then, $f(s,t)$ and $f^*(s,t)$ can be realized by delay-optimum formulae with delay exactly

$$\Big\lceil \log_2\big(W(s) + W(t)\big) \Big\rceil\,.$$

In other words, they can be realized with delay $d \in \mathbb{N}$ if and only if

$$W(s) + W(t) \leq 2^d\,.$$

2.6.3 Adder Optimization Algorithms

Most adder circuits known so far are prefix adders, so we will first introduce this concept before presenting the fastest known prefix adders. In a prefix adder, the inputs are grouped into pairs of consecutive propagate and generate signals as in the following definition. We will see that this is an elegant way to construct adder circuits with a good size, but not all adder circuits are prefix adders. In particular, the delay of any prefix adder is away from the optimum by a factor up to 1.44. Hence, afterwards, we will turn to non-prefix adders. For both types of adders, we will first review adder circuits that optimize circuit depth and later consider the more general case of delay optimization for prescribed arrival times.

Definition 2.6.22. The **adder prefix operator** $\_ \circ_{\mathrm{p}} \_ : \{0,1\}^2 \times \{0,1\}^2 \to \{0,1\}$ is defined by

$$\begin{pmatrix} y_1 \\ x_1 \end{pmatrix} \circ_{\mathrm{p}} \begin{pmatrix} y_0 \\ x_0 \end{pmatrix} = \begin{pmatrix} y_1 \vee (x_1 \wedge y_0) \\ x_1 \wedge x_0 \end{pmatrix}$$

for input pairs $(x_0, y_0), (x_1, y_1) \in \{0,1\}^2$.

A very useful property of the adder prefix operator is its associativity.

Proposition 2.6.23. *The adder prefix operator is associative.*

Proof. For three input pairs $(x_0, y_0), (x_1, y_1), (x_2, y_2) \in \{0,1\}^2$, we have

$$\begin{aligned}
\left(\begin{pmatrix} y_2 \\ x_2 \end{pmatrix} \circ_{\mathrm{p}} \begin{pmatrix} y_1 \\ x_1 \end{pmatrix}\right) \circ_{\mathrm{p}} \begin{pmatrix} y_0 \\ x_0 \end{pmatrix} &= \begin{pmatrix} y_2 \vee (x_2 \wedge y_1) \\ x_2 \wedge x_1 \end{pmatrix} \circ_{\mathrm{p}} \begin{pmatrix} y_0 \\ x_0 \end{pmatrix} \\
&= \begin{pmatrix} \big(y_2 \vee (x_2 \wedge y_1)\big) \vee \big((x_2 \wedge x_1) \wedge y_0\big) \\ (x_2 \wedge x_1) \wedge x_0 \end{pmatrix} \\
&= \begin{pmatrix} y_2 \vee \Big(x_2 \wedge \big(y_1 \vee (x_1 \wedge y_0)\big)\Big) \\ x_2 \wedge (x_1 \wedge x_0) \end{pmatrix} \\
&= \begin{pmatrix} y_2 \\ x_2 \end{pmatrix} \circ_{\mathrm{p}} \begin{pmatrix} y_1 \vee (x_1 \wedge y_0) \\ x_1 \wedge x_0 \end{pmatrix} \\
&= \begin{pmatrix} y_2 \\ x_2 \end{pmatrix} \circ_{\mathrm{p}} \left(\begin{pmatrix} y_1 \\ x_1 \end{pmatrix} \circ_{\mathrm{p}} \begin{pmatrix} y_0 \\ x_0 \end{pmatrix}\right). \qquad \square
\end{aligned}$$

The following way how to use parallel prefix graphs for the construction of adder circuits was described, among others, by Knowles [Kno99]. Assume that we compute an adder circuit on n input pairs $p_0, g_0, \ldots, p_{n-1}, g_{n-1}$. Defining auxiliary variables $z_i = (g_i, p_i) \in \{0,1\}^2$ for all $i = 0, \ldots, n-1$, by Equation (2.18), we have

$$\begin{aligned}
\begin{pmatrix} c_{i+1} \\ p_i \wedge p_{i-1} \wedge \ldots \wedge p_0 \end{pmatrix} &= \begin{pmatrix} g_i \\ p_i \end{pmatrix} \circ_{\mathrm{p}} \begin{pmatrix} g_{i-1} \\ p_{i-1} \end{pmatrix} \circ_{\mathrm{p}} \ldots \circ_{\mathrm{p}} \begin{pmatrix} g_0 \\ p_0 \end{pmatrix} \\
&= z_i \circ_{\mathrm{p}} z_{i-1} \circ_{\mathrm{p}} \ldots \circ_{\mathrm{p}} z_0 \qquad (2.39)
\end{aligned}$$

for every $i = 0, \ldots, n-1$. Hence, an adder circuit on n input pairs can be obtained from a circuit computing $z_i \circ_{\mathrm{p}} z_{i-1} \ldots \circ_{\mathrm{p}} z_0$ for all $i = 0, \ldots, n-1$. As the adder prefix operator is associative by Proposition 2.6.23, any bracketing of the right-hand side of Equation (2.39) will lead to a valid adder circuit. Thus, the following problem is closely related to the ADDER OPTIMIZATION PROBLEM:

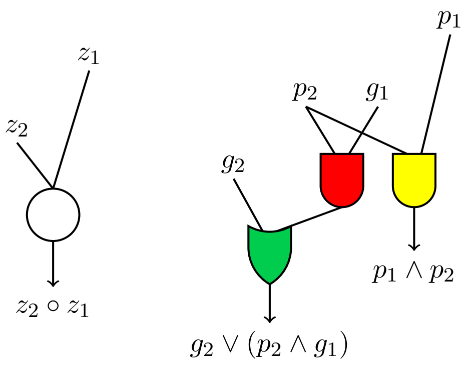


Figure 2.15: Translating an adder prefix gate into 3 logic gates.

Parallel Prefix Problem
Instance: An associative operator $\circ\colon \{0,1\} \times \{0,1\} \to \{0,1\}$, $n \in \mathbb{N}$.
Task: Construct a circuit over $\Omega = \{\circ\}$ on inputs $z_0, \dots z_{n-1}$ computing the prefixes $Z_i := z_i \circ z_{i-1} \circ \dots \circ z_0$ for all $i = 0, \dots, n-1$.

We call an associative operator $\circ\colon \{0,1\} \times \{0,1\} \to \{0,1\}$ a **prefix operator** and a gate representing the prefix operator a **prefix gate**. A circuit solving the Parallel Prefix Problem is called a **parallel prefix graph**, and a circuit computing a single prefix $z_{n-1} \circ z_{n-2} \circ \dots \circ z_0$ is called a **prefix graph**.

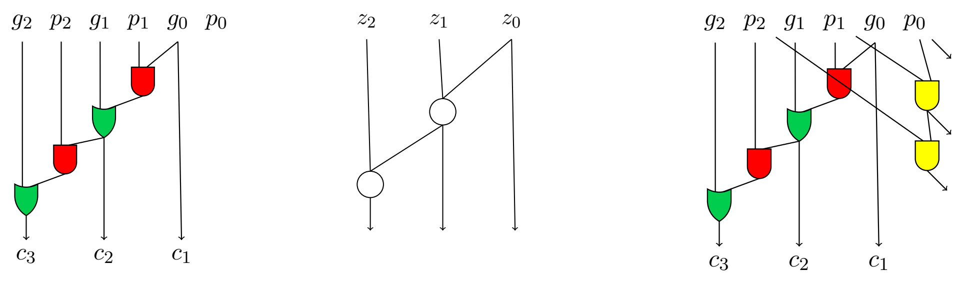
When $\circ = \circ_\mathrm{p}$ is the adder prefix operator from Definition 2.6.22, then a solution for the Parallel Prefix Problem contains an adder circuit, but not every adder circuit can be obtained this way. We call the gates arising from the adder prefix operator **adder prefix gates**. Another special case of the Parallel Prefix Problem is the following problem:

Parallel And-Prefix Problem
Instance: $n \in \mathbb{N}$.
Task: Construct a circuit over $\Omega = \{\text{And2}\}$ on inputs $z_0, \dots z_{n-1}$ computing the And tree $z_i \wedge z_{i-1} \wedge \dots \wedge z_0$ for all $i = 0, \dots, n-1$.

We call a circuit solving the Parallel And-Prefix Problem an **And-prefix circuit**.

In Equation (2.39), we see that for the adder prefix operator $\circ_\mathrm{p}$, a solution to the Parallel Prefix Problem also contains a solution to the Parallel And-Prefix Problem. But of course, with $\circ =$ And, the Parallel And-Prefix Problem on its own is also a special case of the Parallel Prefix Problem.

An adder circuit that arises from a parallel prefix graph with $\circ = \circ_\mathrm{p}$ by using Equation (2.39) and then mapping back each adder prefix gate to And and Or gates as in Figure 2.15 is called a **prefix adder**. We also call And and Or gates **logic gates** and a graph consisting of logic gates a **logic graph** in order to highlight the contrast to the parallel prefix graph consisting of prefix gates. In Figure 2.15, we see that transforming a parallel prefix graph into a logic graph means replacing each prefix gate by 3 logic gates. Hence, a prefix-gate size of n yields a logic-gate size of $3n$, and a prefix-gate depth of d yields a logic-gate depth between d and $2d$.


g_2 p_2 g_1 p_1 g_0 p_0
c_3 c_2 c_1
z_2 z_1 z_0
g_2 p_2 g_1 p_1 g_0 p_0
c_3 c_2 c_1

(a) The ripple-carry adder on $n = 3$ input pairs.

(b) The serial prefix graph on $n = 3$ inputs.

(c) Logic graph for Figure 2.16(b).

Figure 2.16: A ripple-carry adder and its corresponding prefix graph, the serial prefix graph.

Name	Depth	Size	Max. Fanout
Sklansky [Skl60]	$\log_2 n$	$\frac{1}{2}n\log_2 n$	$\frac{1}{2}n + 1$
Ofman [Ofm62]	$2\log_2 n - 1$	$3n - \log_2 n - 2$	$\frac{1}{2}\log_2 n$
Kogge and Stone [KS73]	$\log_2 n$	$n\log_2 n - \frac{n}{2}$	2
Ladner and Fischer [LF80]	$\log_2 n + f$	$2\left(1 + 2^{-f}\right)n$	$\geq n \cdot 2^{-f-1} + 1$
Brent and Kung [BK82]	$2\log_2 n - 1$	$\frac{1}{2}(5n - \log_2 n - 8)$	2

Table 2.1: Overview of parallel prefix graphs for n inputs, where n is a power of two. Depth, size and fanout are measured in the prefix graph, not in the corresponding logic graph.

For our overview of prefix adders, we shall assume that n is a power of two.

First, note that the ripple-carry adder introduced in Section 2.4 is a prefix adder corresponding to the **serial prefix graph** (see Zimmermann [Zim98]) which consists of a single path of prefix gates with depth $n - 1$ and size $n - 1$. Figure 2.16(b) depicts the serial prefix graph for $n = 3$ corresponding to the ripple-carry adder in Figure 2.16(a). Figure 2.16(c) shows the graph that arises from transforming each prefix gate in Figure 2.16(b) into logic gates as in Figure 2.15. In this special case, the yellow AND gates are not needed for the actual adder circuit in Figure 2.16(a). While the size of the ripple-carry adder is optimum, its depth is far away from the lower bound $\lceil \log_2 n \rceil$ on the depth of any binary circuit. As the prefix operator is associative by assumption, each prefix can be computed with optimum depth $\lceil \log_2 n \rceil$ using Huffman coding (see Theorem 2.3.21). However, if each prefix is computed separately using Remark 2.4.7 with this method, the size of the arising circuit is $\mathcal{O}(n^2)$. This method is also called the **unoptimized tree-prefix algorithm** [Zim98].

In the previous decades, there have been numerous attempts to find parallel prefix graphs with a better trade-off between depth and size than these two extreme constructions. Table 2.1 summarizes the most important works. The first prefix graph with logarithmic depth and linear size was introduced by Ofman [Ofm62]. In practice, the construction by Kogge and Stone [KS73] is widely used as it has optimum prefix-gate depth and a maximum fanout of 2. See [Zim98] for an overview on these and other constructions for parallel prefix graphs. We especially note the method by Ladner and Fischer [LF80] which we will use as a sub-routine in Chapter 8.

Theorem 2.6.24 (Ladner and Fischer [LF80])**.** *Let $\circ$ be any binary associative operator. For any $n \in \mathbb{N}_{>0}$ and any constant $0 \leq f \leq \lceil \log_2 n \rceil$, the* Parallel Prefix Problem *can be solved with depth at most $\lceil \log_2 n \rceil + f$ and size at most $2\left(1 + 2^{-f}\right)n$.*

In particular, for any $0 \leq f \leq \lceil \log_2 n \rceil$, there is a circuit S_n^f solving the Parallel And-Prefix Problem *with $\operatorname{depth}\left(S_n^f\right) \leq \lceil \log_2 n \rceil + f$ and $\operatorname{size}\left(S_n^f\right) \leq 2\left(1 + 2^{-f}\right)n$.*

Furthermore, for any $0 \leq f \leq \lceil \log_2 n \rceil$, there is a circuit solving the Adder Optimization Problem *and the* Parallel And-Prefix Problem *on n input pairs at the same time. For the adder circuit, the resulting depth is at most $2(\lceil \log_2 n \rceil + f)$, while for the* And*-prefix circuit, it is at most $\lceil \log_2 n \rceil + f$. The total size of the circuit is bounded by $6\left(1 + 2^{-f}\right)n$.*

The Ladner-Fischer circuits for the parallel prefix problem are constructed by induction on n. Following the detailed size analysis by Wegener [Weg87], it is not hard to show that the number of steps for the construction of the circuit for n inputs and $0 \leq f \leq \lceil \log_2 n \rceil$ is bounded by $2\left(1 + 2^{-f}\right)n$. This implies the following statement.

Proposition 2.6.25. *Given $n \in \mathbb{N}_{>0}$ and $0 \leq f \leq \lceil \log_2 n \rceil$, the parallel prefix circuits by Ladner and Fischer [LF80] can be computed in time $\mathcal{O}(n)$.* □

In a enumerative approach with heuristic pruning, Roy et al. [Roy+14] evaluate the delay of prefix adders regarding arrival times computed after physical design, but the optimization goal is depth and not delay. In Roy et al. [Roy+15], the authors propose a polynomial-time variant. Choi [Cho04] constructs a prefix adder with optimum prefix delay, but with a quadratic number of gates. Modifications of the adder prefix operator have led to improved constructions for a small numbers of inputs, e.g., the so-called Ling-adders or Jackson adders, see, e.g., a comparison with other adders by Keeter et al. [Kee+11].

In Figure 2.15, we see that the adder prefix operator is unbalanced in the sense that in its corresponding logic graph, the inputs g_1 and p_2 traverse 2 logic gates, while the inputs g_2 and p_1 only traverse 1 logic gate. Thus, there are also prefix adders that optimize logic-gate depth (or logic-gate delay) directly in order to overcome the problem that the transition from prefix gates to logic gates might double the depth.

Assume that input arrival times $a(p_0), a(g_0), \ldots, a(p_{n-1}), a(g_{n-1}) \in \mathbb{N}$ are given. In adder circuit optimization, most works assume that $a(p_i) = a(g_i)$ for all $i \in \{0, \ldots, n-1\}$. The following works use the value $V = \sum_{i=0}^{n-1} 2^{a(g_i)}$ to estimate the delay of their circuits which is similar as our instance weight from Definition 2.3.16. See also Remark 2.6.27 for a comparison.

Rautenbach, Szegedy, and Werber [RSW07] provide a prefix adder with logic-gate delay $2 \log_2 V + 6 \log_2 \log_2 n + \mathcal{O}(1)$ using at most $\mathcal{O}(n \log_2 \log_2 n)$ logic gates. Held and Spirkl [HS17b] improved this result to a logic-gate delay of $1.441 \log_2 V + 5 \log_2 \log_2 n + 4.5$ and a size of at most $6n \log_2 \log_2 n$ logic gates. Both variants have a maximum fanout of at least $\sqrt{n}$.

Held and Spirkl [HS17b] give a lower bound of

$$\log_\varphi\left(\sum_{i=0}^{n-1}\varphi^{a(g_i)}\right)-1 \tag{2.40}$$

on the delay of AND-OR path circuits on n input pairs constructed via prefix graphs (thus also of prefix adders), where $\varphi = \frac{1+\sqrt{5}}{2} \approx 1.618$ is the golden ratio. Consequently, even in the case of depth optimization, any prefix adder (in particular any adder from Table 2.1) has a logic-gate depth of at least $1.44\log_2 n - 1$.

Thus, we now consider non-prefix adders. A non-prefix adder with depth $\log_2 n + 7\sqrt{2\log_2 n} + 14$ and size $9n$ has been proposed by Khrapchenko [Khr67]. Gashkov, Grinchuk, and Sergeev [GGS07] provided improvements for concrete small values of n. For arbitrarily large n, but still only for n that are a power of 2, the construction has been improved by Held and Spirkl [HS17a] to a depth of

$$\log_2 n + 8\left\lceil\sqrt{\log_2 n}\right\rceil + 6\left\lceil\log_2\left\lceil\sqrt{\log_2 n}\right\rceil\right\rceil + 2$$

and size of at most $13.5n$ (even at most $9.5n$ if $n \geq 4096$). In particular, the maximum fanout of the circuits by Held and Spirkl [HS17a] is 2, while it is up to linear for the circuits by Khrapchenko [Khr67]. This is the previously best known upper bound on the depth of a linear-size adder circuit with n input pairs.

Some other algorithms for adder optimization regard input arrival times, but most lack provable guarantees: For adders with general arrival times, there are a greedy heuristic by Yeh and Jen [YJ03] and a dynamic program by Liu et al. [Liu+03], but for both, no theoretical guarantee is shown. Oklobdzija [Okl94], Stelling and Oklobdzija [SO96a], and Stelling and Oklobdzija [SO96b] minimize the delay of adders for certain input arrival time patterns occurring in multiplication units. However, these approaches cannot be extended to arbitrary input arrival times.

Spirkl [Spi14] provides adder circuits for delay optimization regarding n input pairs $p_0, g_0, \ldots, p_{n-1}, g_{n-1}$ with $a(p_i) = a(g_i)$ for each $i \in \{0, \ldots, n-1\}$. Spirkl [Spi14] claims that an upper bound on the delay of her adder circuits is given by

$$\lceil\log_2 V\rceil + 5\sqrt{2\log_2 n} + 2\log_2\log_2 n + 16\,,$$

where again $V = \sum_{i=0}^{n-1} 2^{a(g_i)}$, and that the size of her circuits is at most $11n$. There are known problems with this analysis. Probably, these could be fixed in a way that the depth remains $\log_2 V + \mathcal{O}\left(\sqrt{\log_2 n}\right)$ and the size $\mathcal{O}(n)$, but some constants will increase.

2.6.4 AND-OR Path Optimization Algorithms

The depth optimization problem for AND-OR paths is actually solved from an asymptotic point of view (when size and fanout are not considered):

Theorem 2.6.26 (Grinchuk [Gri08]). *Given $m \in \mathbb{N}$, $m \geq 2$, an* AND-OR *path on $m+1$ inputs can be realized by a circuit C with depth at most*

$$\operatorname{depth}(C) \leq \log_2 m + \log_2\log_2 m + 3\,.$$

The work of Commentz-Walter [Com79] implies that, asymptotically, this depth bound is optimum up to an additive constant, see also Corollary 2.6.8; and Hitzschke [Hit18] estimates this constant to be roughly 8 asymptotically, see also Remark 2.6.9. Grinchuk does not analyze the size or fanout of the circuits arising from his proof. But using that they are formula circuits with a special structure, it is not hard to see that the maximum fanout is bounded by the depth, so logarithmic in m, and by Observation 2.3.6, the size is thus at most $\mathcal{O}(m \log_2 m)$.

The crucial ideas that make Grinchuk's circuits so fast are

(i) the introduction of extended And-Or paths, see Definition 2.6.14, and

(ii) the dualization concept that allows optimizing $g(t)$ as well as $g^*(t)$, see Theorem 2.1.31.

In particular, Item (ii) allows Grinchuk to use recursion formulas for both $g(t)$ and $g(t^*)$, e.g., the alternating split with both an odd (see Corollary 2.6.17) and an even prefix (see Corollary 2.6.18); and Item (i) allows him to apply these splits also to extended And-Or paths.

In [Gri13], Grinchuk provides several other algorithms that allow him to construct circuits with best depths known so far for up to 2000000 inputs. He actually provides several algorithms – an exact algorithm with a running time of $\Omega(4^m)$, together with heuristic modifications that potentially lead to sub-optimum solutions, but allow practical running times. He says that his exact algorithm can only be used for up to twenty or thirty inputs.

The idea of Grinchuk's exact algorithm is to compute the optimum achievable depth for all Boolean functions on m inputs in a bottom-up dynamic program, where each Boolean function is identified by its truth table. Naively, his dynamic programming table thus would have 2^{2^m} entries. Grinchuk's main contribution is the observation that a truth table of size m – called a "passport" in [Gri13] – suffices to identify a monotone And-Or path circuit. This way, he can reduce the table size to 2^m, which implies a running time of $\Omega(4^m)$ to compute all table entries.

The fastest exact algorithm for depth optimization of And-Or paths is due to Hegerfeld [Heg18]. He in fact proposes two enumeration algorithms constructing depth-optimum And-Or path circuits over Ω_{mon} of small length. He can also enumerate monotone And-Or path circuits with non-optimum depth with an increase in – nevertheless exponential – running time, which leads to optimum solutions with respect to delay for certain arrival time profiles. In particular, for up to 19 inputs, he constructs all circuits that are depth-optimum and, among all depth-optimum circuits, size-optimum. Furthermore, for up to 29 inputs, he constructs a depth-optimum formula circuit that has optimum size among all formula circuits C where for each vertex $v \in \mathcal{V}(C)$, the sub-circuit C_v has optimum depth. The running time of this algorithm is $\mathcal{O}\left(\left(\sqrt{6}\right)^m\right)$.

The previously best known approaches for delay optimization of And-Or paths with arbitrary prescribed input arrival times lack the ideas (i) and (ii). Table 2.2 summarizes these results. For each algorithm, we show two upper bounds on the delay of the resulting circuit with respect to different instances as in the following remark.

Remark 2.6.27. Recall that for us, an instance of the And-Or Path Circuit Optimization Problem consists of inputs $t_0, \dots, t_{m-1}$ with arbitrary input arrival

Work	Delay	Size	Max. Fanout
[RSW06]	$1.441 \log_2 V + 3$	$4n - 3$	3
	$1.441 \log_2 W + 3$	$2m + 1$	3
[HS17b]	$1.441 \log_2 V + 2.674$	$3n - 3$	2
	$1.441 \log_2 W + 2.674$	$1.5m$	2
[RSW03],	$(1+\varepsilon)\lceil \log_2 V \rceil + \frac{3}{\varepsilon} + 5$	$(n-1)\big(\frac{1}{\varepsilon} + 2\big)$	2
[Spi14]	$(1+\varepsilon)\lceil \log_2 W \rceil + \frac{3}{\varepsilon} + 5$	$\frac{m}{2}\big(\frac{1}{\varepsilon} + 2\big)$	2
[RSW03],	$(1+\varepsilon)\lceil \log_2 V \rceil + \frac{3}{\varepsilon} + 5$	$6n - 6$	$2^{\frac{1}{\varepsilon}}$
[Spi14]	$(1+\varepsilon)\lceil \log_2 W \rceil + \frac{3}{\varepsilon} + 5$	$3m$	$2^{\frac{1}{\varepsilon}}$
[Spi14]	$\lceil \log_2 V \rceil + 2\sqrt{2 \log_2 n} + 6$	$(n-1)\Big(\sqrt{2 \log_2 n} + 3\Big)$	2
	$\lceil \log_2 W \rceil + 2\sqrt{2 \log_2 m - 1} + 6$	$\frac{m}{2}\Big(\sqrt{2 \log_2 m - 1} + 3\Big)$	2
[Spi14]	$\lceil \log_2 V \rceil + 2\sqrt{2 \log_2 n} + 6$	$6n - 6$	$2^{\sqrt{2 \log_2 n}} + 1$
	$\lceil \log_2 W \rceil + 2\sqrt{2 \log_2 m - 1} + 6$	$3m$	$2^{\sqrt{2 \log_2 m - 1}} + 1$

Table 2.2: The previously best known algorithms for the AND-OR PATH CIRCUIT OPTIMIZATION PROBLEM on inputs $t_0, \dots, t_{m-1}$ with arrival times $a(t_0), \dots, a(t_{m-1}) \in \mathbb{N}$. For each method, the first line states the original delay bound, assuming $m = 2n$ and $a(t_{2i}) = a(t_{2i+1})$ for all $i \in \{0, \dots, n-1\}$ and $V = \sum_{i=0}^{n-1} 2^{a(t_{2i})}$; and the second line considers arbitrary m, arbitrary arrival times and $W = \sum_{i=0}^{m-1} 2^{a(t_i)}$.

times $a(t_0), \dots, a(t_{m-1}) \in \mathbb{N}$. However, based on the relation to adder circuits, all works mentioned in Table 2.2 assume that m is even and $a(t_{2i}) = a(t_{2i+1})$ for all $i \in \{0, \dots, n-1\}$. Furthermore, they use $V = \sum_{i=0}^{n-1} 2^{a(t_{2i})}$ to formulate their delay bounds, while we use $W = \sum_{i=0}^{m-1} 2^{a(t_i)}$ from Definition 2.3.16. Hence, for instances with $a(t_{2i}) = a(t_{2i+1})$, we have $V = \frac{W}{2}$. But if we use an adder circuit which assumes that $a(t_{2i}) = a(t_{2i+1})$ on an instance where this is not fulfilled, for each $i = 0, \dots, n-1$, we need to set both $a(t_{2i})$ and $a(t_{2i+1})$ to their maximum value. In the worst case, we thus have $V = W$. Moreover, if m is odd, we need to add an artificial input.

Hence, when stating upper bounds on the delay for the approaches in Table 2.2 on our more general instances, we need to assume that $V = W$ and $n = \lceil \frac{m}{2} \rceil$.

As one of the applications of our AND-OR path optimization algorithms is optimizing a critical path on computer chips (see Chapter 7), our algorithms allow an arbitrary number of inputs and arbitrary arrival times. Hence, by the above remark, our algorithms have a natural practical advantage over the algorithms from Table 2.2 on such instances.

The circuits presented in Table 2.2 have a linear or almost linear size, but, when restricted to depth optimization, i.e., when $W = m$, a significantly worse depth bound than the circuits by Grinchuk [Gri08]. For the sake of a better comparison, we describe the algorithms by Rautenbach, Szegedy, and Werber [RSW06] and Held and Spirkl [HS17b] in detail using our own notation as they are the only algorithms from Table 2.2 that have been implemented in practice.

Assume for simplicity that $m = 2n$ is even. Rautenbach, Szegedy, and Werber

[RSW06] run a dynamic program for the computation of AND-OR paths of type $g(x)$ (i.e., ending with an AND gate), for given Boolean inputs $x = (x_0, \dots, x_{2n-1})$. The table entries are the AND-OR paths $g(t)$ on consecutive subsets t of x. For each table entry, the best realization that can be obtained using the alternating split

$$g(t) = g\big((t_0, t_1, \dots, t_{k-1})\big) \vee \Big(\operatorname{sym}\big((t_0, t_2, \dots, t_{k-2})\big) \wedge g\big((t_k, \dots, t_{r-1})\big)\Big) \tag{2.41}$$

explained in Corollary 2.6.19 for some even $k \in \mathbb{N}$ with $2 \leq k < m$ is chosen. Here, the symmetric tree $\operatorname{sym}\big((t_0, t_2, \dots, t_{k-2})\big)$ is not realized via Huffman coding (see Theorem 2.3.21), but recursively, following the same scheme as the recursion for the computation of the AND-OR paths. This way, the authors can save gates and obtain a linear size, but their delay bound is by a factor of up to 1.441 away from the lower bound of $\lceil \log_2 W \rceil$ (cf. Equation (2.40)). Note that the dynamic program table has n^2 entries, explaining the running time of $\mathcal{O}(n^3)$ since there are $\mathcal{O}(n)$ possibilities to choose the split position in (2.41).

Held and Spirkl [HS17b] consider the computation of the dual AND-OR path function $g^*(t)$ and apply the alternating split from Lemma 2.6.13 i.e.,

$$g^*(t) = g^*\big((t_0, \dots, t_{k-1})\big) \vee \Big(\operatorname{sym}\big((t_1, t_3, \dots, t_k)\big) \wedge g^*\big((t_{k+1}, \dots, t_{r-1})\big)\Big) \tag{2.42}$$

for odd $k \in \mathbb{N}$ with $1 \leq k < m$. Again, the symmetric trees are built according to the alternating split. Instead of running a dynamic program to compute the optimum solution with respect to these restructuring options, the authors choose the optimum splitting options directly by analyzing so-called Fibonacci trees. This leads to a running time of $\mathcal{O}(n \log_2 n)$, where the assumption is made that all additions take constant time. The size and delay improvement compared to [RSW06] are due to a slightly more careful construction and analysis. As we shall make use of this result in our constructions and comparisons, we highlight it in the following theorem. Recall that by Remark 2.6.27, our assumptions on the input differs from those in [HS17b], so the bounds stated here differ slightly from the original bounds shown in Table 2.2.

Theorem 2.6.28 (Held and Spirkl [HS17b])**.** *Given m inputs $t = (t_0, \dots, t_{m-1})$, a circuit for the* AND-OR *path $f(t)$ with depth at most $1.441 \log_2 W + 2.674$, size at most $1.5m$ and maximum fanout 2 can be constructed in time $\mathcal{O}(m \log_2 m)$.*

In Chapter 8, we will use this algorithm to construct all carry bits of an adder circuit in the special case of depth optimization:

Corollary 2.6.29. *For any $n \in \mathbb{N}$, there is an adder circuit on n input pairs with depth at most $1.441 \log_2 n + 2.674$ and a size of at most $\frac{3}{2}(n^2 - n)$.*

Proof. We apply Remark 2.4.7 with the carry-bit computation method from Held and Spirkl [HS17b], see Theorem 2.6.28, in order to obtain an adder circuit. Its depth is $1.441 \log_2 n + 2.674$, while its size is at most

$$\sum_{i=1}^{n} (3i - 3) = 3 \sum_{i=1}^{n} i - 3n = 3\frac{n(n+1)}{2} - 3n = \frac{3}{2}\Big(n^2 - n\Big). \qquad \square$$

It is no coincidence that [RSW06] and [HS17b] achieve a very similar delay bound of roughly $1.441 \log_2 V$: Both circuit constructions can be viewed as optimum prefix graphs as defined in Section 2.6.3: Using the adder prefix operator

$$\begin{pmatrix} y_1 \\ x_1 \end{pmatrix} \circ_{\mathrm{p}} \begin{pmatrix} y_0 \\ x_0 \end{pmatrix} = \begin{pmatrix} y_1 \vee (x_1 \wedge y_0) \\ x_1 \wedge x_0 \end{pmatrix},$$

from Definition 2.6.22, we have

$$\begin{aligned}&\begin{pmatrix} g^*\big((t_0,\dots,t_{k-1})\big) \\ t_1 \wedge t_3 \wedge \dots \wedge t_k \end{pmatrix} \circ_{\mathrm{p}} \begin{pmatrix} g^*\big((t_{k+1},\dots,t_{r-1})\big) \\ t_{k+2} \wedge t_{k+4} \wedge \dots \wedge t_{r-1} \end{pmatrix} \\ =& \begin{pmatrix} g^*\big((t_0,\dots,t_{k-1})\big) \vee \big((t_1 \wedge t_3 \wedge \dots \wedge t_k) \wedge g^*\big((t_{k+1},\dots,t_{r-1})\big)\big) \\ t_1 \wedge t_3 \wedge \dots \wedge t_{r-1} \end{pmatrix},\end{aligned}$$

so computing an optimum prefix graph is in fact the same idea as computing $g^*(t)$ best possible via splits of type (2.42). Similarly, the alternating split in (2.41) can be expressed using the adder prefix operator.

As shown in Equation (2.40), Held and Spirkl [HS17b] proved that the delay of an AND-OR path circuit derived from a prefix graph will deviate from the optimum by a factor of 1.44 in the worst case. Moreover, they show that the delays of the AND-OR path circuits by Held and Spirkl [HS17b] and Rautenbach, Szegedy, and Werber [RSW06] are only by an additive margin of 5 away from the optimum logic-gate delay of any prefix graph. Hence, in order to obtain AND-OR path circuits which are closer to the lower bounds, we need to consider non-prefix circuits.

Rautenbach, Szegedy, and Werber [RSW03] presented AND-OR path circuits on n input pairs with delay at most $(1+\varepsilon)\lceil\log_2 V\rceil + c_\varepsilon$ (for any $\varepsilon > 0$), where c_ε is a number depending on ε only. Spirkl [Spi14] specified the delay bound to $(1+\varepsilon)\lceil\log_2 V\rceil + \frac{6}{\varepsilon} + 8 + 5\varepsilon$ and improved it to $(1+\varepsilon)\lceil\log_2 V\rceil + \frac{3}{\varepsilon} + 5$. Note that Table 2.2 gives two result rows with this delay, but with different sizes and fanouts that can be traded off.

Moreover, Spirkl [Spi14] described non-prefix circuits with a delay of at most

$$\lceil\log_2 V\rceil + 2\sqrt{2\log_2 n} + 6 \tag{2.43}$$

and sizes and fanouts as in the table, where again there are two variants. For any $\varepsilon > 0$, this is actually a better delay bound than $(1+\varepsilon)\lceil\log_2 V\rceil + \frac{3}{\varepsilon} + 5$: The latter function assumes its minimum for $\varepsilon = \sqrt{\frac{3}{\lceil\log_2 V\rceil}}$, as can be seen by computing its derivative. Thus, we have

$$\varepsilon\lceil\log_2 V\rceil + \frac{3}{\varepsilon} + 5 \geq 2\sqrt{3\log_2 V} + 5 \geq 2\sqrt{3\log_2 n} + 5 \geq 2\sqrt{2\log_2 n} + 6\,.$$

Note that for this choice of ε, the circuits from rows 5-6 in the table also outperform the circuits from rows 3-4 in the table when considering size and fanout. Any of these four circuits can be constructed in running time $\mathcal{O}(n\log_2 n)$. Up to now, the delay stated in Equation (2.43) obtained by [Spi14] was the fastest delay known for AND-OR path circuits with non-uniform input arrival times.

Chapter 3
Improved Bounds for Depth Optimization

In this section, we consider the depth optimization problem for And-Or paths.

And-Or path Circuit Depth Optimization Problem
Instance: $m \in \mathbb{N}$.
Task: Compute a circuit over $\Omega_{\text{mon}} = \{\text{And}, \text{Or}\}$ realizing an And-Or path on m inputs with minimum possible depth.

We will describe an algorithm that constructs the currently fastest known circuits for And-Or paths with respect to depth, whose idea is largely based on Grinchuk [Gri08]. The main result proven in [Gri08] is the following theorem.

Theorem 2.6.26 (Grinchuk [Gri08])**.** *Given $m \in \mathbb{N}$, $m \geq 2$, an* And-Or *path on $m+1$ inputs can be realized by a circuit C with depth at most*

$$\operatorname{depth}(C) \leq \log_2 m + \log_2 \log_2 m + 3\,.$$

Grinchuk [Gri08] focuses on the existence result covered by this theorem. He proves in an algorithmic fashion that a realization with the claimed depth exists, but he does not explicitly state the algorithm or analyze the size of the arising circuits. In Section 3.1, we will present a modified algorithm with running time $\mathcal{O}(m \log_2 m)$ that allows us to improve the additive constant in Theorem 2.6.26 by roughly 1.5 in Section 3.2. However, the main advantage of our algorithm over Grinchuk's is that we can prove in Section 3.4 – based on new symmetric tree constructions in Section 3.3 – that the arising circuits have a size linear in the number of inputs. Our circuits are hence the first circuits known that fulfill the best possible asymptotic depth bound (cf. Commentz-Walter [Com79], Corollary 2.6.7) and have a linear size.

Remark. The notation for (extended) And-Or paths used by Grinchuk [Gri08] differs from ours: We always consider an And-Or path $f(t) = f\big((t_0, \dots, t_{m-1})\big)$ with m inputs, while Grinchuk analyzes an And-Or path with $m+1$ inputs, i.e., the And-Or path $f(t) = f\big((t_0, \dots, t_m)\big)$. This notational difference needs to be taken into account when comparing the depth bounds. As a consequence, we adapt the definitions inherited from Grinchuk's proof to our notation.

Note that the depth bound claimed in [Gri08] is actually by 1 better than the bound stated in Theorem 2.6.26. But there appears to be a gap in the proof of

Lemma 7 in [Gri08] for which it is unknown whether it can be closed. The upper bound stated in Theorem 2.6.26 follows easily from the existing part of the proof of Grinchuk's Lemma 7.

3.1 Algorithm

The depth optimization algorithm by Grinchuk [Gri08] is a recursive approach using some of the recursion strategies presented in Section 2.6.2 that work on extended AND-OR paths rather than on AND-OR paths only. Instead of estimating the depth of a good circuit realizing $f\big((s_0, \dots, s_{n-1}), (t_0, \dots, t_{m-1})\big)$ for given $n, m \in \mathbb{N}$, Grinchuk proves Theorem 2.6.26 the following way: He fixes a depth bound $d \in \mathbb{N}$ and a number of symmetric inputs n and determines up to which number of alternating inputs m extended AND-OR paths can be realized by a circuit with depth d. From this, he derives the claimed depth bound. For this approach, the following definition is essential.

Definition 3.1.1 (Grinchuk [Gri08])**.** Given $d, n \in \mathbb{N}$, the *capacity* of d and n is

$$m(d,n) := \max\Big\{m \in \mathbb{N} : \text{There is a circuit for } f\big((s_0, \dots, s_{n-1}), (t_0, \dots, t_{m-1})\big) \\ \text{with depth at most } d\,.\Big\},$$

where $m(d,n) := -\infty$ in case there is no such m.

Note that we also allow $m(d,n) = 0$ in Definition 3.1.1. Furthermore, note that by Corollary 2.5.3, we have

$$m(d,n) = \max\Big\{m \in \mathbb{N} : \text{There is a circuit for } f^*\big((s_0, \dots, s_{n-1}), (t_0, \dots, t_{m-1})\big) \\ \text{with depth at most } d\,.\Big\}.$$

Finding an exact formula for $m(d,n)$ for general $d, n \in \mathbb{N}$ is an open problem, but we will give a lower bound on $m(d,n)$ in Proposition 3.1.14. For this, we first state basic properties about the capacity and compute it exactly for small values of d and n.

Observation 3.1.2 (Grinchuk [Gri08])**.** For $d, n \in \mathbb{N}$, we have $m(d+1, n) \geq m(d,n)$ and $m(d,n) \leq m(d, n-1)$.

Lemma 3.1.3. *For $d, n \in \mathbb{N}$, we have $m(d,n) \in \mathbb{N}$ if and only if $n \leq 2^d$.*

Proof. When $n \leq 2^d$, by Observation 2.6.21, the function $f\big((s_0, \dots, s_{n-1}), ()\big)$ is symmetric and can be realized with depth d. Hence, we have $m(d,n) \geq 0$ for $n \leq 2^d$. Vice versa, if there is some $m \in \mathbb{N}$ such that a realization for $f\big((s_0, \dots, s_{n-1}), (t_0, \dots, t_{m-1})\big)$ with depth at most d exists, then the lower bound from Proposition 2.6.1 implies $n \leq 2^d$. □

For small values of d and n, the capacity can be computed easily.

Lemma 3.1.4 (Grinchuk [Gri08])**.** *We have:*

$$\begin{array}{lllll} m(0,0) = 1 & m(0,1) = 0 & & & \\ m(1,0) = 2 & m(1,1) = 1 & m(1,2) = 0 & & \\ m(2,0) = 3 & m(2,1) = 3 & m(2,2) = 2 & m(2,3) = 1 & m(2,4) = 0 \end{array}$$

For all $d \in \{0,1,2\}$ and $n \in \mathbb{N}$ with $n > 2^d$, we have $m(d,n) = -\infty$.

Proof. The last statement holds due to Lemma 3.1.3.

For $n, m \in \mathbb{N}$ with $m \leq 2$, Observation 2.6.21 yields a realization of $f((s_0, \dots, s_{n-1}), (t_0, \dots, t_{m-1}))$ with delay $\lceil \log_2(m+n) \rceil$. This implies $m(0,0) \geq 1$, $m(0,1) \geq 0$, $m(1,0) \geq 2$, $m(1,1) \geq 1$, $m(1,2) \geq 0$, $m(2,2) \geq 2$, $m(2,3) \geq 1$ and $m(2,4) \geq 0$. The realization $f\big((), (t_0, t_1, t_2)\big) = t_0 \wedge (t_1 \vee t_2)$ implies $m(2,0) \geq 3$, and the realization $f\big((s_0), (t_0, t_1, t_2)\big) = (s_0 \wedge t_0) \wedge (t_1 \vee t_2)$ implies $m(2,1) \geq 3$.

Now we shall see that $m(d,n)$ does not exceed the computed lower bounds. By Proposition 2.6.1, the depth of $f\big((s,t)\big)$ is at least $\lceil \log_2(n+m) \rceil$. This implies $m(0,0) \leq 1$, $m(0,1) \leq 0$, $m(1,0) \leq 2$, $m(1,1) \leq 1$, $m(1,2) \leq 0$, $m(2,1) \leq 3$, $m(2,2) \leq 2$, $m(2,3) \leq 1$ and $m(2,4) \leq 0$. From Observation 2.6.10, we conclude that $m(2,0) = 3$. □

For larger values of d and n, Grinchuk [Gri08] gives a lower bound on $m(d,n)$ in two steps: First, he bounds $m(d,n)$ from below by a recursively defined function $M(d,n)$, where each $M(d,n)$ is an even natural number. The function $M(d,n)$ is directly connected to Grinchuk's recursion formulas. Secondly, he gives a numerical lower bound on the values $M(d,n)$ from which he can derive this depth bound. We proceed differently: We skip the intermediate step of defining $M(d,n)$ and directly give a numerical lower bound on $m(d,n)$ which is marginally stronger than Grinchuk's bound. This leads to a slightly better depth bound and simplifies the size analysis of the arising circuit. In order to be able to prove a linear size bound in Corollary 3.4.21, we also use slightly different recursion formulas than Grinchuk in our algorithm.

Definition 3.1.5. We define the function $\mu\colon \mathbb{N}_{>0} \times \mathbb{N} \to \mathbb{R}$ by

$$\mu(d,n) := \frac{2^d - n - 2}{d} + 2\,.$$

Unlike Grinchuk's approach, in this work, $\mu(d,n)$ will not always be a lower bound on $m(d,n)$, but $\lfloor \mu(d,n) \rfloor$ will, as we will prove in Proposition 3.1.14.

Observation 3.1.6. Given $d, n, m \in \mathbb{N}$ with $d \geq 1$, we have $m \leq \mu(d,n)$ if and only if $n \leq 2^d - d(m-2) - 2$.

We will develop an algorithm (see Algorithm 3.1) that, given m and n, computes a circuit for an extended And-Or path with n symmetric and m alternating inputs with depth at most d, where d is minimum with $m \leq \mu(d,n)$.

Definition 3.1.7. Given $n, m \in \mathbb{N}$ we define

$$d_{\min}(n,m) := \min\big\{\, d \in \mathbb{N}_{>0} : m \leq \mu(d,n) \,\big\}\,.$$

As $\lim_{d \to \infty} \mu(d,n) = \infty$, the value $d_{\min}(n,m)$ is well-defined.

Lemma 3.1.8. *Table 3.1 shows the value $d_{\min}(n,m)$ for all $1 \leq m \leq 9$ and $0 \leq n \leq 11$.*

Proof. Given $n, m \in \mathbb{N}$ and $d = d_{\min}(n,m)$, we have $m \leq \mu(d,n)$. Hence, Observation 3.1.6 implies the following inequalities:

- For $d = 1$, we have $n \leq 2 - 1(m-2) - 2 = 2 - m$.
- For $d = 2$, we have $n \leq 4 - 2(m-2) - 2 = 6 - 2m$.
- For $d = 3$, we have $n \leq 8 - 3(m-2) - 2 = 12 - 3m$.

m \ n	0	1	2	3	4	5	6	7	8	9	10	11
1	1	1	2	2	2	3	3	3	3	3	4	4
2	1	2	2	3	3	3	3	4	4	4	4	4
3	2	3	3	3	4	4	4	4	4	4	4	5
4	3	4	4	4	4	4	4	5	5	5	5	5
5	4	4	4	5	5	5	5	5	5	5	5	5
6	5	5	5	5	5	5	5	5	5	5	5	6
7	5	5	5	5	5	5	6	6	6	6	6	6
8	5	6	6	6	6	6	6	6	6	6	6	6
9	6	6	6	6	6	6	6	6	6	6	6	6

Table 3.1: The value $d_{\text{min}}(n, m)$ for all $3 \leq m \leq 9$ and $0 \leq n \leq 11$ as calculated using Lemma 3.1.8. Cells are colored by the containing number $d_{\text{min}}(n, m)$.

- For $d = 4$, we have $n \leq 16 - 4(m-2) - 2 = 22 - 4m$.
- For $d = 5$, we have $n \leq 32 - 5(m-2) - 2 = 40 - 5m$.
- For $d = 6$, we have $n \leq 64 - 6(m-2) - 2 = 74 - 6m$.

From these statements, together with the minimum choice of d, the table follows. □

We need some basic properties about $\mu(d, n)$ before being able to state our depth optimization algorithm.

Lemma 3.1.9. *Given $d, n \in \mathbb{N}$ with $d \geq 2$, we have $\mu(d, n) < 2^d$.*

Proof. The function $x \mapsto 2^x(x-1) - 2x + 2$ is strictly monotonely increasing for all $x \geq 2$ as its derivative is

$$\ln(2)2^x(x-1) + 2^x - 2 = 2^x\big(\ln(2)x - \ln(2) + 1\big) - 2\,,$$

which is positive for $x \geq 2$. Hence, we have

$$\mu(d, n) = \frac{2^d - n - 2}{d} + 2 \leq \frac{2^d - 2}{d} + 2 \overset{d \geq 2}{<} 2^d\,,$$

where the last inequality can be seen directly for $d = 2$, and holds for $d \geq 3$ by monotonicity of the aforementioned function. □

Lemma 3.1.10. *For $d, n \in \mathbb{N}$ with $d \geq 1$ and $n < 2^d$, we have $\mu(d, n) \geq 1$.*

Proof. We have

$$\mu(d, n) = \frac{2^d - n - 2}{d} + 2 \overset{n \leq 2^d - 1}{\geq} -\frac{1}{d} + 2 \overset{d \geq 1}{\geq} 1\,. \qquad \square$$

Lemma 3.1.11. *Given d, n, m with $2 \leq m \leq \mu(d, n)$, we have $n \leq 2^d - 2$.*

Proof. By Observation 3.1.6, we have $n \leq 2^d - d(m-2) - 2 \overset{m \geq 2}{\leq} 2^d - 2$. □

The next two lemmas give concrete realizations for $f(s, t)$ when either the number m of alternating inputs is small or the expected depth is small.

Lemma 3.1.12. *Let integers $d, n, m \in \mathbb{N}$ with $d \geq 1$, $0 \leq n < 2^d$ and $m \leq \mu(d, n)$ be given. Then, for $m \leq 2$, there is a circuit for $f((s_0, \ldots, s_{n-1}), (t_0, \ldots, t_{m-1}))$ with depth at most d.*

Proof. Under the assumptions of this lemma, we have

$$\begin{aligned}
m + n \leq \frac{2^d - n - 2}{d} + 2 + n \quad &= \quad \frac{2^d + (d-1)n - 2}{d} + 2 \\
&\overset{\substack{n \leq 2^d - 1, \\ d \geq 1}}{\leq} \quad \frac{2^d + (d-1)(2^d - 1) - 2}{d} + 2 \\
&= \quad \frac{d2^d - d - 1}{d} + 2 \\
&= \quad 2^d - \frac{1}{d} + 1 \\
&\overset{d > 0}{<} \quad 2^d + 1\,,
\end{aligned}$$

and as both $m + n$ and 2^d are natural numbers, we even have $m + n \leq 2^d$. For $m \leq 2$, by Observation 2.6.21, this implies that $f(s,t)$ is a symmetric tree that can be realized with depth d. □

Lemma 3.1.13. *Let integers $d, n, m \in \mathbb{N}$ with $1 \leq d \leq 3$, $0 \leq n < 2^d$, and $m \leq \mu(d, n)$ be given. Then, there is a circuit for $f((s_0, \ldots, s_{n-1}), (t_0, \ldots, t_{m-1}))$ with depth at most d.*

Proof. Lemma 3.1.12 proves the statement in the case that $m \leq 2$, so assume that $m \geq 3$. Note that it suffices to show the lemma for $d = d_{\min}(n, m)$. From Table 3.1, we can read off the values of n and m for which we need to verify that a circuit for $f(s,t)$ with depth d exists.

For $d = 1$, by Table 3.1, there is no $m \geq 3$ fulfilling the conditions of this lemma.

For $d = 2$, Table 3.1 and $m \geq 3$ imply $m = 3$ and $n = 0$, and the standard realization of $f(s,t) = g(t)$ has depth $m - 1 = 2$.

For $d = 3$, Table 3.1 and $m \geq 3$ imply $m = 3$ and $n \in \{1, 2, 3\}$ or $m = 4$ and $n = 0$. For $m = 3$, note that the definition of extended And-Or paths (cf. Definition 2.6.14) implies

$$f(s,t) = \operatorname{sym}\big((s_0, \ldots, s_{n-1}, t_0, t_1 \vee t_2)\big)\,. \tag{3.1}$$

This is an instance of the Symmetric Function Delay Optimization Problem with arrival times being 0 for the first $n + 1$ inputs and arrival time 1 for the last input, i.e., an instance with weight $n + 1 + 2 = n + 3 \leq 6$. An optimum symmetric tree on this instance with delay 3 can be constructed e.g., via Huffman coding (cf. Theorem 2.3.21), which implies that the realization (3.1) yields depth 3. For $m = 4$, the standard realization of $f(s,t) = g(t)$ has depth $m - 1 = 3$. □

Finally, we will now see that $\lfloor \mu(d, n) \rfloor$ is a lower bound on $m(d, n)$ when $n < 2^d$.

Proposition 3.1.14. *Consider integers $d, n \in \mathbb{N}$ with $d \geq 1$ and $0 \leq n < 2^d$. For all Boolean input variables $s = (s_0, \ldots, s_{n-1})$ and $t = (t_0, \ldots, t_{m-1})$ with $m \leq \mu(d, n)$, there is a circuit for $f(s,t)$ with depth at most d. In other words, we have $m(d, n) \geq \lfloor \mu(d, n) \rfloor$.*

Proof. We prove the statement by induction on d.

Case 1: Base case. Assume that $d \leq 3$.

In this case, Lemma 3.1.13 proves the statement.

Case 2: Induction step. Assume that the proposition is true for some $d \geq 3$ and all $0 \leq n < 2^d$. Given $d, n, m \in \mathbb{N}$ with $0 \leq n < 2^{d+1}$ and

$$m \leq \mu(d+1, n) \tag{3.2}$$

and input variables $s = (s_0, \dots, s_{n-1})$ and $t = (t_0, \dots, t_{m-1})$, we need to find a circuit for $f(s,t)$ with depth at most $d+1$.

Case 2.1: Assume that $m \leq \mu(d,n)$.

By induction hypothesis, there is a circuit for $f(s,t)$ with depth $d \leq d+1$.

Case 2.2: Assume that $m \leq 2$.

In this case, a circuit for $f(s,t)$ with depth $d+1$ is provided by Lemma 3.1.12.

Case 2.3: Assume that

$$m > \mu(d,n) \tag{3.3}$$

and

$$m \geq 3\,. \tag{3.4}$$

Case 2.3.1: Assume that $n \geq 2^d$.

In this case, we use the symmetric split

$$f(s,t) = \operatorname{sym}(s') \wedge f\big(s \backslash s', t\big) \tag{3.5}$$

from Equation (2.37) with $k := 2^d \leq n$ and a sub-tuple $s' = (s_{i_0}, \dots, s_{i_{k-1}})$ of s. For the depth analysis, it is not important how s' is chosen, we can imagine for simplicity that $s' = (s_0, \dots, s_{k-1})$. In fact, we choose s' as in Algorithm 3.4.

In order to show that Equation (3.5) yields depth d, it suffices to show that both $\operatorname{sym}(s')$ and $f\big((s_k, \dots, s_{n-1}), t\big)$ can be realized with depth at most d. The symmetric tree $\operatorname{sym}(s')$ can be realized by a circuit with depth $d = \lceil \log_2 k \rceil$ using Huffman coding (Theorem 2.3.21) as s' has k entries.

As $m \leq \mu(d+1, n)$ by assumption (3.2) and $m \geq 3$ by assumption (3.4), Lemma 3.1.11 implies $n \leq 2^{d+1} - 2$. Hence, we have

$$|s \backslash s'| = n - k \leq 2^{d+1} - 2 - 2^d = 2^d - 2$$

and

$$\begin{aligned} m &\overset{(3.2)}{\leq} \mu(d+1, n) \\ &\overset{\text{Def. 3.1.5}}{=} \frac{2^{d+1} - n - 2}{d+1} + 2 \\ &\overset{k=2^d}{=} \frac{2^d - (n-k) - 2}{d+1} + 2 \\ &\overset{n-k \leq 2^d - 2}{\leq} \frac{2^d - (n-k) - 2}{d} + 2 \\ &= \mu(d, n-k)\,. \end{aligned}$$

By induction hypothesis, we can find a circuit for $f\big(s \backslash s', t\big)$ with depth d. Together, this shows that the split (3.5) yields a circuit of the Boolean function $f(s,t)$ with depth $d+1$.

Case 2.3.2: Assume that $n < 2^d$.

Case 2.3.2.1: Assume that $m \leq \mu(d,0)$.

As $m \leq \mu(d,0)$ and $n < 2^d$, the AND-OR path $g(t) = f((),t)$ can be realized with depth d by induction hypothesis. Since $n < 2^d$, this means that the symmetric split

$$f(s,t) = \mathrm{sym}(s) \wedge g(t)$$

from Equation (2.33) yields depth $d+1$.

Case 2.3.2.2: Assume that $m > \mu(d,0)$.

By Lemma 3.1.10, we have $\mu(d,n) \geq 1$, so we may choose a maximum odd integer k with

$$k \leq \mu(d,n)\,. \tag{3.6}$$

Assumption (3.3) implies that $k < m$. This allows us to apply the alternating split

$$f(s,t) = f\big(s,t'\big) \wedge f^*\Big(\widehat{t'}, t''\Big) \tag{3.7}$$

from Equation (2.28) with k as length of the odd-length prefix $t' = (t_0, \dots, t_{k-1})$. Recall that $\widehat{t'} = (t_1, t_3, \dots, t_{k-2})$ and $t'' = (t_k, \dots, t_{m-1})$. Due to $n < 2^d$ and Equation (3.6), the induction hypothesis allows us to realize $f\big(s,t'\big)$ with depth d. Hence, for proving that Equation (3.7) yields depth $d+1$, it remains to show that $f^*\Big(\widehat{t'}, t''\Big)$ can be realized with depth d. As the number of inputs of $\widehat{t'}$ and t'' is exactly $\frac{k-1}{2}$ and $m-k$, respectively, by induction hypothesis, for this it suffices to show the following claim.

Claim 1. We have $0 \leq \frac{k-1}{2} < 2^d$ and

$$m - k \leq \mu\left(d, \frac{k-1}{2}\right). \tag{3.8}$$

Proof of claim: Since k is odd, we have $\frac{k-1}{2} \in \mathbb{N}$. Moreover, since $d \geq 2$, we have

$$\frac{k-1}{2} \overset{(3.6)}{\leq} \frac{\mu(d,n)-1}{2} \overset{\text{Lem. 3.1.9}}{<} \frac{2^d-1}{2} < 2^d\,.$$

Now it remains to show (*3.8*). Since k is the maximum odd integer fulfilling (3.6), we have $k+2 > \mu(d,n)$ and thus

$$\begin{aligned}
\mu\left(d, \frac{k-1}{2}\right) + k \quad &= \quad \frac{2^d - \frac{k-1}{2} - 2}{d} + 2 + k \\
&= \quad k - \frac{k-1}{2d} + \frac{2^d-2}{d} + 2 \\
&= \quad k\left(1 - \frac{1}{2d}\right) + \frac{2^d-2}{d} + 2 + \frac{1}{2d} \\
&\overset{\substack{d\geq 1,\\ k+2>\mu(d,n)}}{>} \quad (\mu(d,n)-2)\left(1 - \frac{1}{2d}\right) + \frac{2^d-2}{d} + 2 + \frac{1}{2d} \\
&\overset{\text{Def. 3.1.5}}{=} \quad \frac{2^d-n-2}{d}\left(1 - \frac{1}{2d}\right) + \frac{2^d-2}{d} + 2 + \frac{1}{2d} \\
&= \quad \frac{2^d-2}{d}\left(2 - \frac{1}{2d}\right) - \left(1 - \frac{1}{2d}\right)\frac{n}{d} + 2 + \frac{1}{2d}\,.
\end{aligned} \tag{3.9}$$

We will now rewrite the first term of (3.9).

$$\begin{aligned}
\frac{2^d-2}{d}\left(2-\frac{1}{2d}\right) &= \frac{2^{d+1}-4}{d} - \frac{2^d-2}{2d^2} \\
&= \left(1+\frac{1}{d}\right)\cdot\frac{2^{d+1}-4}{d+1} - \frac{2^d-2}{2d^2} \\
&= \frac{2^{d+1}-2}{d+1} - \frac{2}{d+1} + \frac{1}{d}\cdot\frac{2^{d+1}-4}{d+1} - \frac{2^d-2}{2d^2} \\
&= \frac{2^{d+1}-2}{d+1} + \frac{-4d^2+4d2^d-8d-2^d(d+1)+2(d+1)}{2d^2(d+1)} \\
&= \frac{2^{d+1}-2}{d+1} + \frac{(3d-1)2^d-4d^2-6d+2}{2d^2(d+1)}\,. \qquad (3.10)
\end{aligned}$$

From this, we deduce

$$\begin{aligned}
&\mu\left(d,\frac{k-1}{2}\right)+k \\
\overset{(3.9)}{>}\;& \frac{2^d-2}{d}\left(2-\frac{1}{2d}\right) - \left(1-\frac{1}{2d}\right)\frac{n}{d}+2+\frac{1}{2d} \\
\overset{(3.10)}{=}\;& \frac{2^{d+1}-2}{d+1} + \frac{(3d-1)2^d-4d^2-6d+2}{2d^2(d+1)} - \frac{n(2d-1)}{2d^2}+2+\frac{1}{2d} \\
=\;& \frac{2^{d+1}-2}{d+1} + \frac{(3d-1)2^d-4d^2-6d+2-n(2d-1)(d+1)+d(d+1)}{2d^2(d+1)}+2 \\
=\;& \frac{2^{d+1}-2}{d+1} + \frac{(3d-1)2^d-4d^2-6d+2-n(2d^2+d-1)+d^2+d}{2d^2(d+1)}+2 \\
=\;& \frac{2^{d+1}-n-2}{d+1}+2+\frac{(3d-1)2^d-3d^2-5d+2-n(d-1)}{2d^2(d+1)} \\
\overset{\text{Def. 3.1.5}}{=}\;& \mu(d+1,n)+\frac{(3d-1)2^d-3d^2-5d+2-n(d-1)}{2d^2(d+1)} \\
\overset{(3.2)}{\geq}\;& m+\frac{(3d-1)2^d-3d^2-5d+2-n(d-1)}{2d^2(d+1)}\,.
\end{aligned}$$

Thus, in order to prove Equation (*3.8*), it suffices to show that

$$(3d-1)2^d-3d^2-5d+2-n(d-1)\geq 0\,. \qquad (3.11)$$

But we have

$$\begin{aligned}
&(3d-1)2^d-3d^2-5d+2-n(d-1) \\
\overset{n\leq 2^d-1}{\geq}\;& (3d-1)2^d-3d^2-5d+2-(2^d-1)(d-1) \\
=\;& (3d-1)2^d-3d^2-5d+2-2^d(d-1)+d-1 \\
=\;& 2^{d+1}d-3d^2-4d+1 \\
>\;& d(2^{d+1}-3d-4)\,.
\end{aligned}$$

The last term is positive as the function $d\mapsto 2^{d+1}-3d-4$ is strictly monotonely increasing for $d\geq 3$ (as its derivative $d\mapsto \ln(2)2^{d+1}-3$ is positive for all $d\geq 3$) and evaluates to 3 for $d=3$. This proves (*3.11*), (*3.8*) and thus the claim. □

We conclude that realization (3.7) yields a circuit for $f(s,t)$ with depth at most $d+1$ in case 2.3.2.2.

This finishes the proof of the induction step (case 2) and hence of the proposition. □

Algorithm 3.1 (page 74) states the algorithm to compute a circuit for $f(s,t)$ which arises from the proof of Proposition 3.1.14.

Note that we do not explicitly state how the occurring optimum symmetric circuits are constructed. E.g., we could apply Huffman coding [Huf52], see Theorem 2.3.21, and construct each symmetric circuit as a formula circuit on the inputs. Then, the circuit computed by Algorithm 3.1 would be a formula circuit with a size in $\mathcal{O}(m\log_2(m+n)+n)$, see Theorem 3.4.1. As we construct various symmetric trees during Algorithm 3.1, a better idea is to use the output of non-trivial symmetric circuits in multiple symmetric circuits. We shall see in Theorem 3.4.19 that this leads to a size of $\mathcal{O}(m+n)$. As long as we always construct optimum symmetric circuits, this does not make a difference regarding the depth analysis of the arising circuit. Relatedly, we do not specify how the subset s' of s is chosen in line 11 as for the depth analysis, this is irrelevant. Thus, we postpone these topics until the size discussion in Section 3.4.

3.2 Depth Analysis

In order to give an upper bound on the depth of the circuits computed by Algorithm 3.1, we need several technical lemmas.

Lemma 3.2.1. *For $x \in \mathbb{R}$ with $x > e$, the function $\phi(x) = \frac{x}{\ln x}$ is strictly monotonely increasing.*

Proof. The statement can be shown by proving that the first derivative of ϕ is strictly positive for $x > e$. We have

$$\frac{d}{dx}\phi(x) = \frac{\ln x - 1}{\ln^2 x} = \frac{1}{\ln x}\left(1 - \frac{1}{\ln x}\right),$$

and this is positive as both factors of the right-hand side function are positive for $x > e$. □

Lemma 3.2.2. *For $x \geq 2$, the function $\phi(x) = \frac{2^x}{x}$ is strictly monotonely increasing.*

Proof. As $x \geq 2$, we may equivalently show that the function $y \mapsto \frac{y}{\log_2 y}$ is monotonely increasing for $y \geq 4$. This follows from Lemma 3.2.1 because $\frac{y}{\log_2 y} = \ln 2 \frac{y}{\ln y}$ and $\ln 2 > 0$. □

Lemma 3.2.3. *Let $c := 0.58$ and $n, m \in \mathbb{N}$ with $m \geq 3$. Consider the function*

$$\vartheta(n,m) = 2^c(m+n)\log_2 m - (m-2)\big(\log_2(m+n) + \log_2\log_2 m + c\big) - n - 2\,.$$

We have

$$\vartheta(n,m) \geq 0\,.$$

Proof. We first prove the statement for $n = 0$ and then for $n > 0$.

Case 1: Assume that $n = 0$.

Note that $\vartheta(0,m) = 2^c m\log_2 m - (m-2)(\log_2 m + \log_2\log_2 m + c) - 2$.

Case 1.1: Assume that $3 \leq m \leq 7$.

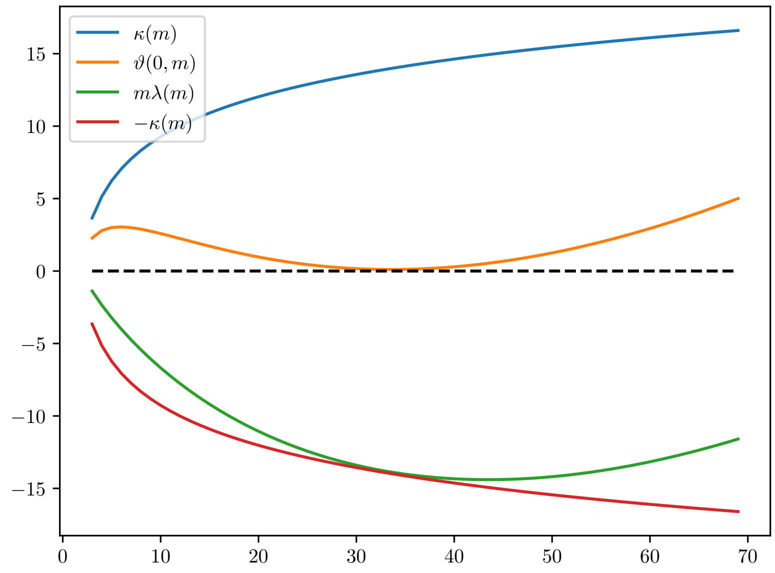


Figure 3.1: The functions $\kappa(m)$, $\vartheta(0,m)$, and $m\lambda(m)$ from the proof of Lemma 3.2.3. Here, we have $\vartheta(0,m) = m\lambda(m) + \kappa(m)$, and we show that $\vartheta(0,m) \geq 0$ for all $m \geq 3$. For a better comparison with $m\lambda(m)$, we also show the function $-\kappa(m)$.

In this cases, we prove the statement by explicitly enumerating all cases:

$$\begin{aligned}
\vartheta(0,3) &= 2^c \cdot 3\log_2 3 - (\log_2 3 + \log_2\log_2 3 + c) - 2 &&> 2 > 0\\
\vartheta(0,4) &= 2^c \cdot 4 \cdot 2 - 2(2 + 1 + c) - 2 &&> 2 > 0\\
\vartheta(0,5) &= 2^c \cdot 5\log_2 5 - 3(\log_2 5 + \log_2\log_2 5 + c) - 2 &&> 3 > 0\\
\vartheta(0,6) &= 2^c \cdot 6\log_2 6 - 4(\log_2 6 + \log_2\log_2 6 + c) - 2 &&> 3 > 0\\
\vartheta(0,7) &= 2^c \cdot 7\log_2 7 - 5(\log_2 7 + \log_2\log_2 7 + c) - 2 &&> 2 > 0
\end{aligned}$$

Case 1.2: Assume that $m \geq 8$.
Writing

$$\lambda(m) := (2^c - 1)\log_2 m - \log_2\log_2 m - c$$

and

$$\kappa(m) := 2(\log_2 m + \log_2\log_2 m + c) - 2\,,$$

we have

$$\vartheta(0,m) = m\lambda(m) + \kappa(m)\,. \tag{3.12}$$

Figure 3.1 depicts the functions $\vartheta(0,m)$, $\kappa(m)$ and $m\lambda(m)$; and $\lambda(m)$ is plotted in Figure 3.2. We will examine these functions in a series of claims.

Claim 1. The function $\lambda(m)$ is monotonely increasing in m for $m \geq 8$.
Proof of claim: The derivative of $\lambda(m)$ is

$$\frac{d}{dm}\lambda(m) = \frac{2^c - 1}{\ln(2)m} - \frac{1}{\ln(2)m\ln m} = \frac{1}{\ln(2)m}\left(2^c - 1 - \frac{1}{\ln m}\right). \tag{3.13}$$

Since $\ln(2)m > 0$, we have $\frac{d}{dm}\lambda(m) > 0$ if and only if

$$2^c - 1 - \frac{1}{\ln m} > 0\,,$$

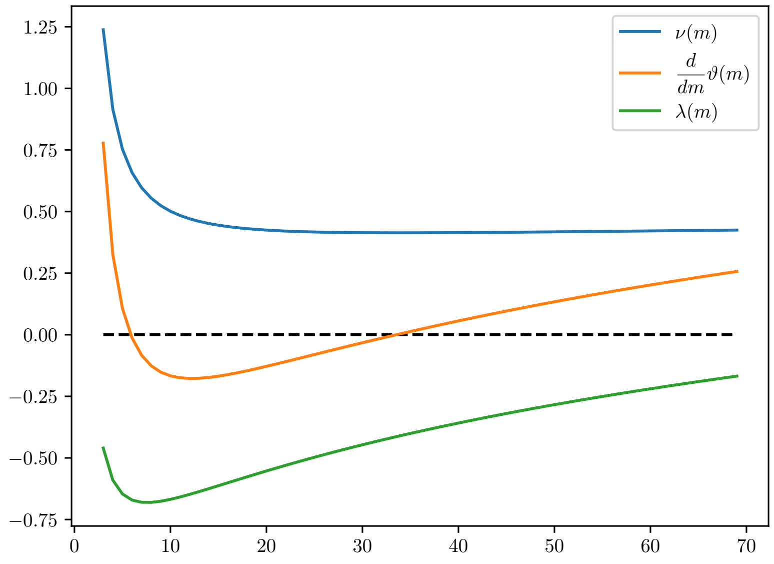


Figure 3.2: The functions $\nu(m)$, $\frac{d}{dm}\vartheta(0,m)$, and $\lambda(m)$ from the proof of Lemma 3.2.3. Here, we have $\frac{d}{dm}\vartheta(0,m) = \lambda(m) + \nu(m)$.

which is fulfilled for $m \geq 8$ since $c \geq 0.57$. Thus, $\lambda(m)$ is monotonely increasing in m for $m \geq 8$. □

Based on the computations in Claim 1, we can compute the derivative of the function $m\lambda(m)$:

$$\frac{d}{dm}\big(m\lambda(m)\big) = \lambda(m) + m\frac{d}{dm}\lambda(m) \overset{(3.13)}{=} \lambda(m) + \frac{1}{\ln(2)}\left(2^c - 1 - \frac{1}{\ln m}\right) \tag{3.14}$$

We compute the derivatives of $\kappa(m)$ and $\vartheta(0,m)$:

$$\begin{aligned} \frac{d}{dm}\kappa(m) &= 2\left(\frac{1}{\ln(2)m} + \frac{1}{\ln(2)m\ln m}\right) \qquad (3.15)\\ \frac{d}{dm}\vartheta(0,m) &\overset{(3.12)}{=} \frac{d}{dm}(m\lambda(m)) + \frac{d}{dm}\kappa(m) \\ &\overset{(3.14),\,(3.15)}{=} \lambda(m) + \frac{1}{\ln(2)}\left(2^c - 1 - \frac{1}{\ln m}\right) + \frac{2}{\ln(2)m} + \frac{2}{\ln(2)m\ln m} \\ &= \lambda(m) + \frac{2^c-1}{\ln(2)} + \frac{-m + 2\ln m + 2}{\ln(2)m\ln m} \end{aligned}$$

Writing

$$\nu(m) := \frac{2^c-1}{\ln(2)} + \frac{-m + 2\ln m + 2}{\ln(2)m\ln m},$$

we have

$$\frac{d}{dm}\vartheta(0,m) = \lambda(m) + \nu(m).$$

Figure 3.2 depicts these three functions.

Claim 2. Let $m \geq 8$. The function $\nu(m)$ is monotonely decreasing for $m \leq 33$ and monotonely increasing for $m \geq 33$.

Proof of claim: We compute the derivative of $\nu(m)$:

$$\begin{aligned}\frac{d}{dm}\nu(m) &= \frac{\left(-1+\frac{2}{m}\right)\ln(2)m\ln m - (-m+2\ln m+2)\ln(2)(\ln m+1)}{\ln^2(2)m^2\ln^2 m}\\ &= \frac{\left(-1+\frac{2}{m}\right)m\ln m - (-m+2\ln m+2)(\ln m+1)}{\ln(2)m^2\ln^2 m}\\ &= \frac{-m\ln m+2\ln m+m\ln m-2\ln^2 m-2\ln m+m-2\ln m-2}{\ln(2)m^2\ln^2 m}\\ &= \frac{m-2\ln^2 m-2\ln m-2}{\ln(2)m^2\ln^2 m}\end{aligned}$$

Note that $\nu(m)$ is monotonely increasing if and only if the function $\widetilde{\nu}(m) := m-2\ln^2 m-2\ln m-2$ fulfills $\widetilde{\nu}(m)\geq 0$. In order to see when this is the case, we compute the derivative of $\widetilde{\nu}(m)$:

$$\frac{d}{dm}\widetilde{\nu}(m) = 1-\frac{4\ln m}{m}-\frac{2}{m}$$

Note that each summand of $\frac{d}{dm}\widetilde{\nu}(m)$ is monotonely increasing (for $-\frac{4\ln m}{m}$, this follows from Lemma 3.2.1 and $m\geq 8>e$), hence so is $\frac{d}{dm}\widetilde{\nu}(m)$. From this, by plugging in $m=11,12$ in $\frac{d}{dm}\widetilde{\nu}(m)$, we deduce

$$\frac{d}{dm}\widetilde{\nu}(m)\begin{cases}<0 & \text{for } 8\leq m\leq 11\,,\\ >0 & \text{for } m\geq 12\,.\end{cases}$$

In other words, the function $\widetilde{\nu}(m)$ is decreasing for $8\leq m\leq 11$ and increasing for $m\geq 12$. Evaluating $\widetilde{\nu}(m)$ for $m\in\{8,33,34\}$, we see that for $m\geq 8$, $m\in\mathbb{N}$, we have $\widetilde{\nu}(m)\geq 0$ if and only if $m\geq 34$. Hence, for $m\geq 8$, the function $\nu(m)$ is monotonely decreasing in the range $8\leq m\leq 33$ and monotonely increasing for $34\leq m$. □

Claim 3. Let $m\geq 8$. The function $\vartheta(0,m)$ is monotonely decreasing for $m\leq 33$ and monotonely increasing for $m\geq 34$.

Proof of claim: We make a case distinction based on whether m is at most 33.

Case 1.2.1: Assume that $8\leq m\leq 33$.

In this case, Claim 2 implies that $\nu(m)$ is monotonely decreasing. Furthermore, by Claim 1, the function $\lambda(m)$ is monotonely increasing in m.

These two statements imply that for $8\leq m\leq 19$, we have

$$\frac{d}{dm}\vartheta(0,m) = \lambda(m)+\nu(m)\leq\lambda(19)+\nu(8) < -0.56+0.56 = 0\,,$$

for $20\leq m\leq 32$, we have

$$\frac{d}{dm}\vartheta(0,m) = \lambda(m)+\nu(m)\leq\lambda(32)+\nu(20) < -0.427+0.425<0\,,$$

and for $m=33$, we have

$$\frac{d}{dm}\vartheta(0,m) = \lambda(33)+\nu(33) < -0.418+0.414<0\,.$$

Case 1.2.2: Assume that $m\geq 34$.

In this case, Claims 1 and 2 imply that both $\lambda(m)$ and $\nu(m)$ are monotonely increasing functions, respectively. Here, we thus have

$$\frac{d}{dm}\vartheta(0,m) = \lambda(m) + \nu(m) \geq \lambda(34) + \nu(34) > -0.41 + 0.41 = 0\,.$$

This proves Claim 3. □

From Claim 3, we deduce that

$$\vartheta(0,m) \geq \vartheta(0,33) > 0.1 > 0 \quad \text{for } 8 \leq m \leq 33$$

and

$$\vartheta(0,m) \geq \vartheta(0,34) > 0.1 > 0 \quad \text{for } m \geq 34\,.$$

This proves this lemma in the case that $n = 0$.

Case 2: Assume that $n > 0$.

Here, we show that

$$\frac{d}{dn}\vartheta(n,m) \geq 0 \tag{3.16}$$

for all $n \geq 0$, $m \geq 3$, which implies that for $n > 0$, we have $\vartheta(n,m) \geq \vartheta(0,m) \overset{\text{case 1}}{\geq} 0$. We compute

$$\frac{d}{dn}\vartheta(n,m) = 2^c \log_2 m - \frac{m-2}{\ln(2)(m+n)} - 1 =: \kappa(n,m)\,.$$

Since $\kappa(n,m) \geq \kappa(0,m)$ for all $n, m \in \mathbb{N}$, it suffices to show $\kappa(0,m) \geq 0$ for all $m \geq 3$ in order to prove Equation (*3.16*). But $\kappa(0,3) = 2^c \log_2 3 - \frac{1}{3\ln(2)} - 1 > 0$ for $c \geq 0$, and for $m \geq 4$, we have

$$\begin{aligned}
\kappa(0,m) &= 2^c \log_2 m - \frac{m-2}{\ln(2)m} - 1 \\
&= 2^c \log_2 m + \frac{1}{\ln(2)}\left(\frac{2}{m} - 1\right) - 1 \\
&\overset{m>0}{\geq} 2^c \log_2 m - \frac{1}{\ln(2)} - 1 \\
&\overset{m\geq 4}{\geq} 2^c \log_2 4 - \frac{1}{\ln(2)} - 1 \\
&\overset{c\geq 0.29}{>} 0\,.
\end{aligned}$$

This proves Equation (*3.16*) and thus the lemma in the case that $n > 0$. □

Finally, we are set to analyze the depth of the circuits computed by Algorithm 3.1.

Theorem 3.2.4. *Given input variables $s = (s_0, \ldots, s_{n-1})$ and $t = (t_0, \ldots, t_{m-1})$ with $m + n > 0$, Algorithm 3.1 computes a circuit $\mathrm{C}(s,t)$ for the Boolean function $f(s,t)$ with depth*

$$\operatorname{depth}(\mathrm{C}(s,t)) = \lceil \log_2(m+n) \rceil$$

for $m \leq 2$ and

$$\operatorname{depth}(\mathrm{C}(s,t)) \leq \log_2(m+n) + \log_2\log_2 m + 1.58$$

for $m \geq 3$.

Algorithm 3.1: Depth optimization for extended And-Or paths

```
Input: Symmetric inputs $s = (s_0, \dots, s_{n-1})$ and alternating inputs
       $t = (t_0, \dots, t_{m-1})$.
Output: Circuit $\mathrm{C}(s,t)$ computing $f(s,t)$.
1  if $m \leq 2$ then
2      return Optimum circuit for $\mathrm{sym}\big((s_0, \dots, s_{n-1}, t_0, \dots, t_{m-1})\big)$.
3  Let $d := d_{\min}(n,m)$.                                  // Hence, $n < 2^d$.
4  if $d \leq 3$ then                                          // Hence, $m \in \{3,4\}$.
5      if $m = 3$ then
6          return Optimum circuit for $\mathrm{sym}\big((s_0, \dots, s_{n-1}, t_0, t_1 \vee t_2)\big)$.
7      if $m = 4$ then                                         // Hence, $n = 0$.
8          return Standard circuit for $g(t)$.
9  else if $n \geq 2^{d-1}$ then
10     Choose $k := 2^{d-1}$.
11     Choose $s' \subseteq s$ with $|s'| = k$.
12     Compute an optimum circuit $S'$ for $\mathrm{sym}(s')$.
13     return $S' \wedge \mathrm{C}\big(s \backslash s', t\big)$.
14 else
15     if $m \leq \mu(d-1, 0)$ then
16         Compute an optimum circuit $S$ for $\mathrm{sym}\big((s_0, \dots, s_{n-1})\big)$.
17         return $S \wedge \mathrm{C}\big((), t\big)$.
18     Choose $1 \leq k < m$ maximum with $k$ odd and $k \leq \mu(d-1, n)$.
19     Set $t' := (t_0, \dots, t_{k-1})$ and $t'' := t \backslash t'$.
20     return $\mathrm{C}(s, t') \wedge \left(\mathrm{C}\left(\widehat{t'}, t''\right)\right)^*$.
```

Proof. For $m \leq 2$, Algorithm 3.1 computes an optimum circuit of $f(s,t)$ with depth $\lceil \log_2(m+n) \rceil$ in line 2.

For $m \geq 3$, we shall see that the algorithm always computes a realization with depth at most $d := d_{\min}(n,m)$, cf. line 3. Since $m \geq 2$, we have $n < 2^d$ by Lemma 3.1.11. Hence, Proposition 3.1.14 yields a circuit for $f(s,t)$ with depth d. The rest of Algorithm 3.1 performs the same steps as the proof of Proposition 3.1.14, hence computes a realization of $f(s,t)$ with depth at most d.

Now let

$$\tilde{d} := \lfloor \log_2(m+n) + \log_2 \log_2 m + 1.58 \rfloor .$$

If we can show that

$$m \leq \mu\left(\tilde{d}, n\right), \tag{3.17}$$

then, by definition of $d = d_{\min}(n,m)$, we have $d \leq \tilde{d}$ and the theorem is proven.

Recall that $\mu\left(\tilde{d}, n\right) = \frac{2^{\tilde{d}} - n - 2}{\tilde{d}} + 2$. By Lemma 3.2.2, for fixed n, the function $x \mapsto \frac{2^x - n - 2}{x} + 2$ is monotonely increasing in x for $x \geq 2$. As $\tilde{d} \geq \log_2(m+n) + \log_2 \log_2 m + 0.58 \overset{m \geq 3}{\geq} 2$, for proving Equation (3.17), it hence suffices to show

$$m \leq \frac{2^{\log_2(m+n) + \log_2 \log_2 m + 0.58} - n - 2}{\log_2(m+n) + \log_2 \log_2 m + 0.58} + 2 . \tag{3.18}$$

The right-hand side of (*3.18*) can be simplified as

$$\frac{2^{\log_2(m+n)+\log_2\log_2 m+0.58}-n-2}{\log_2(m+n)+\log_2\log_2 m+0.58}+2=\frac{2^{0.58}(m+n)\log_2 m-n-2}{\log_2(m+n)+\log_2\log_2 m+0.58}+2\,.$$

Hence, (*3.18*) is implied if we can prove that for $n, m \in \mathbb{N}$ with $m \geq 3$ and $c := 0.58$, we have

$$2^c(m+n)\log_2 m-(m-2)\big(\log_2(m+n)+\log_2\log_2 m+c\big)-n-2\geq 0\,.$$

This is precisely the statement of Lemma 3.2.3. Hence, Equation (*3.18*) is fulfilled and the circuit computed by Algorithm 3.1 has depth at most $\tilde{d}$. □

For And-Or paths, Theorem 3.2.4 yields the following delay guarantee.

Corollary 3.2.5. *Given input variables* $t = (t_0, \dots, t_{m-1})$ *with* $m \geq 2$*, Algorithm 3.1 computes a circuit for* $g(t)$ *with depth at most*

$$\log_2 m+\log_2\log_2 m+1.58\,.$$

This is the same asymptotic depth bound as for the circuits by Grinchuk [Gri08], see also Theorem 2.6.26, but the additive constant is slightly better. An improvement by more than a constant is not possible due to the matching lower bound given by Commentz-Walter [Com79], see also Corollary 2.6.8. As Hitzschke [Hit18] made these lower bounds more precise, see Remark 2.6.9, our circuits are optimum up to an additive constant of roughly 6.58.

3.3 Leftist Circuits and Triangular Sets

This section is a preparation for Section 3.4, where we will show that the circuits constructed in Algorithm 3.1 can be implemented with linear size. As we construct numerous symmetric trees during Algorithm 3.1, we cannot effort each them to have a size linear in the number of inputs independently. Hence, the key idea of this construction is that we build two so-called leftist symmetric circuits, one And circuit and one Or circuit, and use these when constructing symmetric And and Or trees in Algorithm 3.1. This way, we can show that the amount of additional gates needed to construct a single symmetric tree is logarithmic in the number of inputs. More details follow in Section 3.4, but here, we will introduce leftist circuits.

Definition 3.3.1. Let $n \in \mathbb{N}$ and a commutative and associative operator $\circ$ be given. A circuit S on inputs $x_0, \dots, x_{n-1}$ over $\Omega = \{\circ\}$ is called **ordered** if

- the underlying undirected graph of S is acyclic and
- for each vertex $v \in \mathcal{V}(S)$, there is an interval $I_v \subseteq \{0, \dots, n-1\}$ such that $\mathcal{I}_v(S) = \{x_i : i \in I_v\}$.

This yields a partial order on $\mathcal{V}(S)$: We say that a vertex $v \in \mathcal{V}(S)$ is **left** of a vertex $w \in \mathcal{V}(S)$ if $I_v \cap I_w = \emptyset$ and $\max\{\, i \in I_v \,\} < \min\{\, i \in I_w \,\}$, and **right** of w if w is left of v. The two predecessors of any gate vertex are related with respect to this partial order. Thus, we may call the predecessors left and right, respectively. We extend the definitions of left and right from vertices to sets of vertices $W_1, W_2 \subseteq \mathcal{V}(S)$ with $W_1 \cap W_2 = \emptyset$: We say that W_1 is left (right) of W_2 if for any pair of vertices $(w_1, w_2) \in W_1 \times W_2$, the vertex w_1 is left (right) of the vertex w_2.

```
Algorithm 3.2: Leftist circuit construction
  Input: A commutative and associative operation
         $\circ\colon \{0,1\} \times \{0,1\} \to \{0,1\}$, $n \in \mathbb{N}$, $n \geq 1$.
  Output: A leftist circuit $S$ on inputs $x = (x_0, \dots, x_{n-1})$ over the basis
          $\Omega = \{\circ\}$.
1 Let $S$ be a circuit on inputs $x = (x_0, \dots, x_{n-1})$ with no gates.
2 Let $i := 0$.
3 while $n \geq 2$ do
4     Let $k \in \mathbb{N}$ be maximum with $k \leq \log_2 n$.
5     Add an ordered full symmetric $\circ$-tree $T_k$ on $x_i, \dots, x_{i+2^k-1}$ to $S$.
6     Set $i := i + 2^k$.
7     Set $n := n - 2^k$.
8 Let $\mathcal{O}(S) := \{v \in \mathcal{V} : |\delta^+(v)| = 0\}$.
9 return $S$
```

Note that if $n = 2^d$ for $d \in \mathbb{N}$, then there is a unique connected ordered circuit with optimum depth d on inputs $x_0, \dots, x_{n-1}$. Otherwise, if $2^{d-1} < n < 2^d$ for some $d \in \mathbb{N}$, we consider the binary decomposition $n = \sum_{k=0}^{\lfloor \log_2 n \rfloor} a_k 2^k$ with $a_k \in \{0,1\}$ of n. We partition the n inputs into groups of 2^k inputs for each k with $a_k = 1$. On each input group, we construct a connected ordered circuit T_k with depth k. This yields an ordered (but unconnected) circuit. Algorithm 3.2 does exactly this, where the T_k are sorted from left to right by increasing k.

Definition 3.3.2. A circuit arising from Algorithm 3.2 is called **leftist**.

Figure 3.3(a) (page 80) depicts a leftist OR tree on $n = 14$ inputs.

Definition 3.3.3. Let $n \in \mathbb{N}$ and a leftist circuit S on inputs $x_0, \dots, x_{n-1}$ be given. A subset $K \subseteq \mathcal{I}(S)$ of the inputs is called **consecutive** if there are $0 \leq a < b \leq n-1$ with $K = \{x_i : a \leq i \leq b\}$.

Definition 3.3.4. Consider $n \in \mathbb{N}$ and a leftist circuit S on inputs $x_0, \dots, x_{n-1}$. Given a subset $K \subseteq \mathcal{I}(S)$, let $B(K, S) \subseteq \mathcal{V}$ be defined as

$$B(K,S) := \left\{ v \in \mathcal{V} : \mathcal{I}_v(S) \subseteq K \text{ and } \mathcal{I}_w(S) \nsubseteq K \ \forall\, w \in \delta^+(v) \right\}.$$

We call the elements of $B(K, S)$ **boundary vertices** of K with respect to S.

In Figure 3.3(a) (page 80), the boundary vertices of K with respect to S are marked blue.

Note that in a leftist circuit S, for every vertex $v \in \mathcal{V}(S)$, the sub-circuit S_v subordinate to v is a full binary tree. Furthermore, given a subset K of the inputs with boundary set $B := B(K, S)$, we have

$$\dot{\bigcup}_{v \in B} \mathcal{I}_v(S) = K\,. \tag{3.19}$$

In particular, there is an ordering $b_0, \dots, b_{|B(K,S)|-1}$ of the boundary vertices such that b_j is left of b_{j+1} for all $j \in \{0, \dots, |B(K, S)| - 1\}$.

Definition 3.3.5. Let $n \in \mathbb{N}$ and a leftist circuit S on inputs $x_0, \dots, x_{n-1}$ be given. Consider a subset $K \subseteq \mathcal{I}(S)$ of the inputs with boundary vertices $B := B(K, S)$. The **boundary tree sequence** of K with respect to S is the sequence $T_0, \dots, T_{|B|-1}$ of binary trees subordinate to the boundary vertices, sorted from left to right. We call K **triangular** if there is some $J \in \{0, \dots, |B|-1\}$ such that the inputs of $T_0, \dots, T_J$ are consecutive, the inputs of $T_{J+1}, \dots, T_{|B|-1}$ are consecutive, and

$$\begin{aligned} \operatorname{depth}(T_j) &< \operatorname{depth}(T_{j+1}) \quad \text{for } 0 \leq j < J-1, \\ \operatorname{depth}(T_j) &> \operatorname{depth}(T_{j+1}) \quad \text{for } J+1 \leq j < |B|-1\,. \end{aligned} \tag{3.20}$$

We call $T_0, \dots, T_J$ the **increasing** part of the tree boundary sequence, and $T_{J+1}, \dots, T_{|B|-1}$ the **decreasing** part of the tree boundary sequence.

Note that for a given set, the boundary tree sequence is unique; but the value J in the definition of a triangular set is not necessarily unique. The consecutive set K in Figure 3.3(a) (page 80) is triangular, where the boundary vertices are marked blue and we can choose $J = 2$ or $J = 3$. The set $N \backslash K$ in Figure 3.3(a) is not triangular as condition (3.20) is not fulfilled for the boundary tree sequence, which contains trees with depths 2, 0, and 0. In Figure 3.4 (page 82), the sets N, K and $N \backslash K$ are all triangular. The following lemma gives some criteria for when a set K is triangular. Later, in Proposition 3.3.13, we shall see a sufficient condition for $N \backslash K$ to be triangular when K is triangular.

Lemma 3.3.6. *Let $n \in \mathbb{N}$ and a leftist circuit S on inputs $x_0, \dots, x_{n-1}$ be given. Then, the following statements hold:*

(i) The empty set is triangular.

(ii) The set $K = \{x_0, \dots, x_{n-1}\}$ is triangular.

(iii) If $\emptyset \neq K \subseteq \mathcal{I}(S)$ is triangular and x_i is the right-most or left-most vertex in K, then $K \backslash \{x_i\}$ is triangular.

(iv) Let $K \subseteq \mathcal{I}(S)$ be a consecutive subset of the inputs. Then, K is triangular.

Proof. For the empty set, the boundary tree sequence is empty, hence the first statement.

If K contains all inputs of S, then the boundary tree sequence consists of exactly the trees of the leftist circuit S. As their depths are strictly decreasing by definition of a leftist circuit, $K = \{x_0, \dots, x_{n-1}\}$ is triangular, hence the second statement.

Now assume that $K \subseteq \mathcal{I}(S)$ is triangular with x_i being its right-most vertex. Let $B := B(K, S)$ and let $T_0, \dots, T_{|B|-1}$ be the boundary tree sequence of K with respect to S. Then, $T_{|B|-1}$ is the unique tree in the sequence which contains x_i. Let d denote the depth of $T_{|B|-1}$.

If $d = 0$, then $K \backslash \{x_i\}$ is certainly triangular.

Thus, assume that $d > 0$. As x_i is the right-most vertex of $T_{|B|-1}$ and $T_{|B|-1}$ is a full binary tree, deleting x_i and all its successors from $T_{|B|-1}$ results in a sequence $T'_0, \dots, T'_{d-1}$ of full binary trees, ordered from left to right. The boundary tree sequence of $K \backslash \{x_i\}$ is given by $T_0, \dots, T_{|B|-2}, T'_0, \dots, T'_{d-1}$. As x_i is the right-most vertex of $T_{|B|-1}$, the inputs of these trees are consecutive, and if the inputs of $T_{|B|-2}$ and $T_{|B|-1}$ are consecutive, the inputs of $T_{|B|-2}$ and T'_0 are also consecutive. As K is triangular and $\operatorname{depth}(T'_j) = d - j - 1$ for all $j \in \{0, \dots, d-1\}$, Equation (3.20) of Definition 3.3.5 is fulfilled for this sequence. Hence, $K \backslash \{x_i\}$ is triangular.

The proof works analogously if x_i is the left-most vertex of K. This proves the third statement.

The fourth statement follows from the second and the third statement as we can obtain any consecutive set K from $\{x_0, \ldots, x_{n-1}\}$ by successively deleting the right-most or left-most vertex. □

Before showing in Theorem 3.3.12 how we can use leftist circuits in order to save gates, we state some basic properties about leftist circuits.

Lemma 3.3.7. *Let S be a leftist circuit on inputs $x_0, \ldots, x_{n-1}$. Then, the circuit S' arising from deleting $x_0, \ldots, x_{n-1}$ (with their successors being the inputs of S') is again a leftist circuit.*

Proof. By definition, S consists of a collection of connected ordered trees T_k of depth k for each k with non-trivial coefficient in the binary decomposition of n. When removing $x_0, \ldots, x_{n-1}$, for every k with $k > 0$, the tree T_k is transformed into a connected ordered tree T'_{k-1} with 2^{k-1} inputs and depth $k-1$. A possible tree T_0 with depth 0 is simply removed. This is exactly the leftist circuit that Algorithm 3.2 produces with the successors of $x_0, \ldots, x_{n-1}$ as inputs. □

Lemma 3.3.8. *Let S be a leftist circuit on inputs $x_0, \ldots, x_{n-1}$ and $K \subseteq \mathcal{I}(S)$ be triangular with $|K| = k$ and $B := B(K, S)$. Furthermore, let S' be the leftist circuit arising from S by deleting the input vertices (see Lemma 3.3.7), and let $K' := \big\{ v \in \mathcal{V}(S) : \operatorname{depth}(v) = 1$ and $w \in K \ \forall\ w \in \delta^-(v) \big\}$. Then, the following statements are fulfilled:*

(i) *We have $|B| \leq k$.*

(ii) *We have $B = B(K', S') \,\dot\cup\, (K \cap B)$.*

(iii) *The set K' is triangular with respect to S'.*

(iv) *For $k \geq 2$, we have $|K \cap B| \leq 2$, and $|K \cap B| \equiv k \mod 2$, and*

$$|K'| = \begin{cases} \frac{k-1}{2} & \text{if } k \text{ odd,} \\ \frac{k}{2} & \text{if } k \text{ even, } |K \cap B| = 0\,, \\ \frac{k-2}{2} & \text{if } k \text{ even, } |K \cap B| = 2\,. \end{cases}$$

Proof. The first two statements follow directly from the definitions.

The third statement follows from the second: The boundary tree sequence for K with respect to S can be transformed into a boundary sequence for K' with respect to S' by deleting trees with depth 0 and removing the vertices with depth 0 for the other trees.

For proving the fourth statement, note that as K is triangular, there are at most 2 boundary trees with depth 0, hence $|K \cap B| \leq 2$. Furthermore, by Equation (3.19), the inputs of the boundary trees form a partition of K, and each boundary tree with depth at least 1 contains an even number of vertices. Hence, we have $|K \cap B| \equiv k \mod 2$. Together with the second statement, this implies the estimation of $|K'|$. □

Lemma 3.3.9. *Consider $k, n \in \mathbb{N}$ with $n \geq k \geq 3$. Let a leftist circuit S on inputs $x_0, \ldots, x_{n-1}$ and a triangular subset $K \subseteq \mathcal{I}(S)$ with $|K| = k$ be given. Then, for $B := B(K, S)$, we have*

$$|B| \leq 2 \log_2 k - 1\,.$$

Proof. As in Lemma 3.3.7, let S' be the leftist circuit arising from S by deleting the input vertices, and let $K' := \{v \in \mathcal{V}(S) : \mathrm{depth}(v) = 1 \text{ and } w \in K \ \forall\, w \in \delta^-(v)\}$. We prove the statement by induction on k.

For $3 \leq k \leq 6$, we show the statement explicitly. We have

$$\begin{aligned}
|B| &\overset{\text{Lem. 3.3.8,}(ii)}{=} |B(K',S')| + |K \cap B| \\
&\overset{\text{Lem. 3.3.8,}(i)}{\leq} |K'| + |K \cap B| \\
&\overset{\text{Lem. 3.3.8,}(iv)}{=} \begin{cases} \frac{|K|-1}{2} + 1 & \text{if } |K| \text{ odd} \\ \frac{|K|}{2} + 0 & \text{if } |K| \text{ even, } |K \cap B| = 0 \\ \frac{|K|-2}{2} + 2 & \text{if } |K| \text{ even, } |K \cap B| = 2 \end{cases} \\
&\overset{k\in\{3,4,5,6\}}{\leq} 2\log_2 k - 1\,.
\end{aligned}$$

Now, we may assume $k \geq 7$. For $k \geq 7$, property (iv) of Lemma 3.3.8 implies $|K'| \geq 3$. As K' is triangular by Item (iii) of Lemma 3.3.8, we may apply the induction hypothesis to the set K' and the circuit S', which yields $|B(K',S')| \leq 2\log_2(|K'|)-1$. This implies

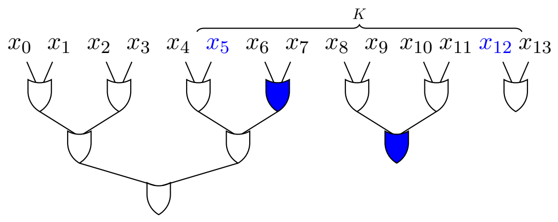
$$\begin{aligned}
|B| &\overset{\text{Lem. 3.3.8,}(ii)}{=} |B(K',S')| + |K \cap B| \\
&\overset{\text{(IH)}}{\leq} 2\log_2\big(|K'|\big) - 1 + |K \cap B| \\
&\overset{\text{Lem. 3.3.8,}(iv)}{=} \begin{cases} 2\log_2\left(\frac{k-1}{2}\right) - 1 + 1 & \text{if } k \text{ odd} \\ 2\log_2\left(\frac{k}{2}\right) - 1 + 0 & \text{if } k \text{ even, } |K \cap B| = 0 \\ 2\log_2\left(\frac{k-2}{2}\right) - 1 + 2 & \text{if } k \text{ even, } |K \cap B| = 2 \end{cases} \\
&\leq 2\log_2 k - 1\,.
\end{aligned}$$

This proves the induction step and hence the lemma. □

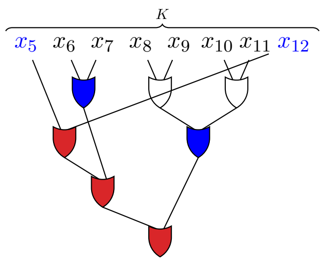
The following theorem states that given a leftist circuit S on all inputs, we can construct an optimum symmetric tree for any triangular subset K of the inputs (plus, possibly, some more inputs) while adding only few gates to S. Figure 3.3(b) shows the constructed tree for S and K from Figure 3.3(a).

Proposition 3.3.10. *Let $n \in \mathbb{N}$ and a leftist circuit S on n inputs $x_0, \ldots, x_{n-1}$ be given. Let $K \subseteq \mathcal{I}(S)\}$ be a triangular set of inputs with boundary vertices $B := B(K,S)$. Furthermore, let inputs L (not necessarily inputs of S) with $K \cap L = \emptyset$ be given. Consider arrival times $a(y) = 0$ for each $y \in K$ and $a(y) \in \mathbb{N}$ for each $y \in L$. A delay-optimum symmetric tree on $K \mathbin{\dot\cup} L$ with respect to arrival times a can be constructed (while possibly reusing the gates in S) using at most $|B| + |L| - 1$ additional gates.*

Proof. Let H denote the circuit arising from Huffman coding (see Theorem 2.3.21) on input set $B \mathbin{\dot\cup} L$, where an input $v \in B$ has arrival time $a(v) = \log_2 |\mathcal{I}_v(S)|$ and an input $v \in L$ has arrival time $a(v)$. Since S is leftist, we have $a(v) \in \mathbb{N}$ for all


K
x_0 x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8 x_9 x_{10} x_{11} x_{12} x_{13}

(a) A leftist OR tree S on $n = 14$ inputs with the set $B = B(K, S)$ of boundary vertices for $K = \{x_5, \ldots, x_{12}\}$ marked blue.



(b) Circuit resulting from Huffman coding on the set B from Figure 3.3(a).

Figure 3.3: Illustration of the proof of Theorem 3.3.12 with $K = \{x_5, \ldots, x_{12}\}$ and $L = \emptyset$. Note that $|B| = 4 = 2\log_2(|K|) - 2$.

Algorithm 3.3: Computation of boundary vertices

```
Input: A leftist circuit $S$ on inputs $x_0, \ldots, x_{n-1}$, a consecutive subset
       $K = (x_a, \ldots, x_b)$ of the inputs, precomputed data from
       Lemma 3.3.11.
Output: The set $B := B(K, S)$.
1 Let $B := \emptyset$.
2 Let $i := a$.
3 while $i \leq b$ do
4     Choose $j$ maximum such that a right ancestor $s_j(x_i)$ of depth $j$ exists
      and $\mathcal{I}_{s_j(x_i)}(S) \subseteq K$.
5     $s := s_j(x_i)$
6     $B := B \cup \{s\}$
7     $i := r(s) + 1$.
8 return $B$.
```

$v \in B$. Thus, the delay of H with respect to arrival times a is exactly

$$\left\lceil \log_2\left(\sum_{v \in B \uplus L} 2^{a(v)}\right)\right\rceil = \left\lceil \log_2\left(\sum_{v \in B} |\mathcal{I}_v(S)| + W(L)\right)\right\rceil \overset{(3.19)}{=} \left\lceil \log_2\big(|K| + W(L)\big)\right\rceil,$$

which is the optimum possible delay of a symmetric tree on inputs K and L by Theorem 2.3.15. Hence, $S \cup H$ contains a delay-optimum symmetric tree on $K \uplus L$. Since H contains exactly $|B| + |L| - 1$ gates, this proves the proposition. □

As stated in Theorem 2.3.21, Van Leeuwen [Lee76] showed that Huffman coding can be performed in linear time if the inputs are sorted by increasing arrival time. Hence, given an input set K which is triangular with respect to a leftist circuit S, once the set $B(K, S)$ of boundary vertices is known, sorted by increasing depth, we can construct an optimum symmetric tree on S (plus a constant number $|L|$ of additional inputs) in time $\mathcal{O}(|B(K, S)|)$.

Hence, we now give an algorithm for determining the boundary vertices in the case that K is consecutive. Here, given a vertex $v \in \mathcal{V}(S)$, we denote the indices of the left-most and right-most input in $\mathcal{I}_v(S)$ by $l(v)$ and $r(v)$, respectively. Furthermore, given an input x_i and a vertex $v \in \mathcal{V}(S)$, we call v a **right ancestor** of x_i if there is a set of vertices $\big\{s_j(x_i) \in \mathcal{V}(S) : j \in \{0, \dots, r\}\big\}$ for some $r \geq 0$ with $s_j(x_i)$ being a left predecessor of $s_{j+1}(x_i)$ for all $j \in \{0, \dots, r-1\}$, $x_i = s_0(x_i)$ and $v = s_r(x_i)$. Note that $s_j(x_i)$ has depth j in S as S is leftist. Let r_i denote the highest depth (and index) of any right ancestor of input x_i.

Given a consecutive subset $K \subseteq \mathcal{I}(S)$ of a leftist circuit S, we use Algorithm 3.3 to determine the boundary vertices. The following lemma shows that the algorithm works correctly in general and estimates its running time.

Lemma 3.3.11. *Let a leftist circuit S on inputs $x_0, \dots, x_{n-1}$ and a consecutive subset $K \subseteq \mathcal{I}(S)$ of size $k := |K|$ be given. Assume that the following data of size $\mathcal{O}(n \log_2 n)$ is available:*

- *For every input x_i and the set $\big\{s_j(x_i)\big\}_{0 \leq j \leq r_i}$ of right ancestors of x_i with depth j, we store a pointer to $s_j(x_i)$ in an array with constant-time access.*
- *For every vertex $v \in \mathcal{V}$, we store the index $r(v)$ of the right-most input in $\mathcal{I}(v)$.*
- *For every $i \in \{0, \dots, n-1\}$, we store the highest depth r_i of any right ancestor of input x_i.*

Then, Algorithm 3.3 computes the set $B := B(S, K)$ of boundary vertices in time $\mathcal{O}(\log_2 k)$. The algorithm can be implemented with the same running time if the output set B has to be sorted by increasing depth.

Proof. As a first step, we prove that Algorithm 3.3 works correctly.

We first see by induction on i that whenever a vertex is added to B, it is a boundary vertex. So consider an iteration of the while-loop (lines 3 to 7) where a vertex $s := s_j(x_i)$ is added to B in line 6. In line 4, j chosen maximum such that a right ancestor $s_j(x_i)$ of depth j exists and $\mathcal{I}_{s_j(x_i)}(S) \subseteq K$. Thus, in order to show that $s := s_j(x_i)$ is a boundary vertex, it remains to show that if s has a successor t, then $\mathcal{I}_t(S) \nsubseteq K$. Assume that s has a successor t and that $\mathcal{I}_t(S) \subseteq K$. By maximality of j, the vertex s must be a right predecessor of t. If $a = i$, then we immediately have $\mathcal{I}_t(S) \nsubseteq K$. If $i > a$, we must have $\mathcal{I}_t(S) \nsubseteq K$ since otherwise, we would have skipped i in the while-loop (lines 3 to 7). Hence, s is a boundary vertex.

Assume now that there is a boundary vertex v that is not added to B by Algorithm 3.3. If in some iteration of the while-loop (lines 3 to 7), we have $i = l(v)$, then v is added to B. So assume that $l(v)$ is skipped, meaning that $x_{l(v)} \in \mathcal{I}_s(S)$ for some s considered in line 5. But two ancestors s and v of an input $x_{l(v)}$ cannot be in B simultaneously by (3.19), so this is a contradiction.

This proves the correctness of Algorithm 3.3.

Now, we analyze its running time. As in each iteration, one vertex is added to B, there are at most B iterations, and by Lemma 3.3.9, we have $B \in \mathcal{O}(\log_2 k)$. The index j computed in line 4 is exactly $j := \min\Big\{\lfloor \log_2(b - i + 1) \rfloor, r_i\Big\}$. Thus, using the precomputed data, each iteration can be implemented to run in constant time, so the total running time of Algorithm 3.3 is $\mathcal{O}(\log_2 k)$.

If, during the algorithm, we use buckets to store the boundary vertices ordered by their depth, the total running time does not increase as there are at most $\lceil \log_2 k \rceil$ possible depths. □

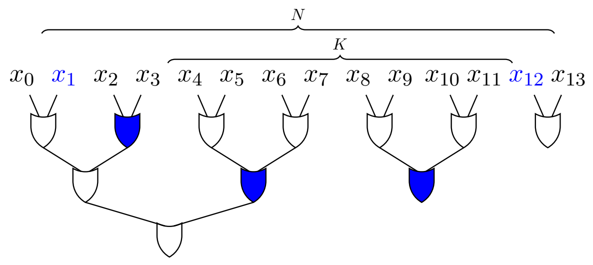

N
K
x_0 x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8 x_9 x_{10} x_{11} x_{12} x_{13}

Figure 3.4: Illustration of Proposition 3.3.13 and Algorithm 3.4. The gates shown form a leftist circuit on inputs $x_0, \dots, x_{13}$. The sets N, K, $N \backslash K$ of inputs are all triangular. The blue vertices are the boundary vertices of N with respect to S.

Indeed, for the leftist circuit S and the consecutive subset K shown in Figure 3.3(a), the algorithm works correctly: The vertices x_i chosen in lines 2 and 7 are x_5, x_6, x_8, and x_{12}, and exactly the blue vertices are added to B.

From the previous statements, we obtain the main theorem of this section.

Theorem 3.3.12. *Let $n \in \mathbb{N}$ and a leftist circuit S on n inputs $x_0, \dots, x_{n-1}$ be given. Let $K \subseteq \mathcal{I}(S)\}$ be a triangular set of inputs with $k := |K|$. Furthermore, let inputs L (not necessarily inputs of S) with $l := |L| \geq 0$ and $K \cap L = \emptyset$ be given. Consider arrival times $a(y) = 0$ for each $y \in K$ and $a(y) \in \mathbb{N}$ for each $y \in L$, where L is sorted by increasing arrival time. A delay-optimum symmetric tree on $K \mathbin{\dot\cup} L$ with respect to arrival times a can be constructed in time $\mathcal{O}(\log_2 k + l)$, assuming the precomputed data from Lemma 3.3.11 is given. The number of additional gates (while possibly reusing the gates in S) is at most $k + l - 1$ for $k \leq 2$ and at most $2\log_2 k + l - 2$ for $k \geq 3$.*

Proof. If $k \leq 2$, we obtain an optimum solution with exactly $k + l - 1$ gates by Huffman coding, see Theorem 2.3.21. Van Leeuwen [Lee76] showed that Huffman coding can be performed in linear time if the inputs are sorted by increasing arrival time, so for constant k, the running time is $\mathcal{O}(l)$.

Now assume that $k \geq 3$. By Proposition 3.3.10, we can compute the delay-optimum symmetric tree on $K \mathbin{\dot\cup} L$ by Huffman coding on the vertices of $B(K, S)$ and L. By Lemma 3.3.9, this set has cardinality $2\log_2 k + l - 2$. From this, the bound on the number of gates follows. By Definition 3.3.5, the inputs belonging to the increasing and decreasing part of the boundary sequence of K are consecutive sets, hence we can apply Lemma 3.3.11 to both to obtain $B(K, S)$ in time $\mathcal{O}(\log_2 k)$, using the precomputed data. Huffman coding on $B(K, S)$ and L can be performed in time $\mathcal{O}(\log_2 k + l)$. □

We will use this theorem for the construction of symmetric circuits during our AND-OR path optimization algorithm Algorithm 3.1 in Section 3.4. Here, we will also need Algorithm 3.4 which, given a triangular set N, extracts a triangular subset K of a certain size such that also $N \backslash K$ is triangular, and such that we can prove in Proposition 3.3.18 that K and $N \backslash K$ cannot be both large simultaneously. Figure 3.4 gives an example for Algorithm 3.4.

Algorithm 3.4: Determining a triangular subset K of a triangular set N

```
Input: A leftist circuit $S$ on inputs $x_0, \dots, x_{r-1}$, a triangular set $N \subseteq \mathcal{I}(S)$
       with $2^{d-1} \leq n < 2^d$ inputs.
Output: A triangular set $K \subseteq N$ with $|K| = 2^{d-1}$.
1 Let $T_0, \dots, T_{|B(N,S)|-1}$ denote the boundary tree sequence for $N$ w.r.t. $S$.
2 if $\exists\ j$ with $\text{depth}(T_j) = d-1$ then
3     return $\mathcal{I}(T_j)$
4 else
5     Set $D := \max\{ d' \in \mathbb{N} : \exists\ j_0 < j_1 \text{ with } \text{depth}(T_{j_0}) = \text{depth}(T_{j_1}) = d' \}$.
6     Choose $j_0 < j_1$ such that $\text{depth}(T_{j_0}) = \text{depth}(T_{j_1}) = D$.
7     return $\bigcup_{j=j_0}^{j_1} \mathcal{I}(T_j)$
```

Proposition 3.3.13. *Let $r \in \mathbb{N}$ and a leftist circuit S on r inputs $x_0, \dots, x_{r-1}$ be given. Consider a triangular subset N of the inputs with cardinality $|N| = n \geq 1$. Let $d \in \mathbb{N}_{\geq 1}$ be the unique integer with $2^{d-1} \leq n < 2^d$, and let $k := 2^{d-1}$. Then, Algorithm 3.4 computes a subset $K \subseteq N$ with $|K| = k$ such that both K and $N \backslash K$ are triangular.*

Proof. We use the notation from Algorithm 3.4.

First, we need to show that the value D in line 5 exists. Thus, assume that there is no tree T_j with depth $d-1$. As $n < 2^d$, the maximum depth of any tree T_j is hence $d-2$. By Definition 3.3.5, the boundary tree sequence contains at most 2 full binary trees with the same number $2^{d'}$ of inputs for each $d' \in \{0, \dots, d-2\}$. Based on this, the value D in line 5 exists as otherwise, there is at most one tree per depth, and we obtain the contradiction

$$2^{d-1} \leq n \leq \sum_{d'=0}^{d-2} 2^{d'} = 2^{d-1} - 1 < 2^{d-1}.$$

Thus, Algorithm 3.4 correctly computes a set $K \subseteq N$.

It is easy to see that K and $N \backslash K$ are triangular with respect to S as $T_{j_0}, \dots, T_{j_1}$ is the boundary tree sequence for K and $T_0, \dots, T_{j_0-1}, T_{j_1+1}, \dots, T_{|B(N,S)-1|}$ is the boundary tree sequence for $N \backslash K$ with increasing part $T_0, \dots, T_{j_0-1}$ and decreasing part $T_{j_1+1}, \dots, T_{|B(N,S)-1|}$.

It remains to show that $|K| = 2^{d-1}$. If K is constructed in line 3, this is certainly the case, so assume that K is constructed in line 7 as $K = \bigcup_{j=j_0}^{j_1} \mathcal{I}(T_j)$. For every $i \in \{D, \dots, d-2\}$, the sequence $T_0, \dots, T_{|B(N,S)|-1}$ must contain a tree with depth i as otherwise, there is $I > D$ such that no tree with depth I exist, which implies

$$\begin{aligned}
2^{d-1} &\leq n \\
&\leq 2\sum_{i=0}^{D} 2^i + \sum_{i=D+1}^{d-2} 2^i - 2^I \\
&= \sum_{i=0}^{d-2} 2^i + \sum_{i=0}^{D} 2^i - 2^I \\
&\leq 2^{d-1} + 2^{D+1} - 2^I - 2 \\
&\overset{I>D}{<} 2^{d-1},
\end{aligned}$$

a contradiction. From this, we conclude that

$$k = \sum_{j=j_0}^{j_1} |\mathcal{I}(T_j)| = 2 \cdot 2^D + \sum_{i=D+1}^{d-2} 2^i = 2^{D+1} + 2^{d-1} - 1 - (2^{D+1} - 1) = 2^{d-1}\,. \quad \square$$

Corollary 3.3.14. *Let $r \in \mathbb{N}$ and a leftist circuit S on r inputs $x_0, \dots, x_{r-1}$ be given. Consider a triangular subset N of the inputs with cardinality $|N| = n \geq 1$, and another input vertex x_i which is directly to the right of N such that $N \cup \{x_i\}$ is triangular. Let $K \subseteq N$ be the triangular set computed by Algorithm 3.4. Then, $(N \backslash K) \cup \{x_i\}$ is triangular.*

Proof. By the proof of Proposition 3.3.13, the set $(N \backslash K)$ is triangular with boundary tree sequence $T_0, \dots, T_{j_0-1}, T_{j_1+1}, \dots, T_{|B(N,S)-1|}$, with increasing part $T_0, \dots, T_{j_0-1}$ and decreasing part $T_{j_1+1}, \dots, T_{|B(N,S)-1|}$.

As the inputs of $T_{j_1+1}, \dots, T_{|B(N,S)-1|}$ and x_i are consecutive, adding x_i to $(N \backslash K)$ results in a new tree with depth d', where d' is minimum such that there is no tree among $T_{j_1+1}, \dots, T_{|B(N,S)-1|}$ with depth d', and all trees with smaller depth are removed from the boundary tree sequence (note that $d' = 0$ may also occur). This new boundary tree sequence fulfills Equation (3.20) of Definition 3.3.5, hence $(N \backslash K) \cup \{x_i\}$ is triangular. $\square$

We will see in Proposition 3.3.18 that either the number of trees in the boundary tree sequence of K is small or $N \backslash K$ is small. For estimating what "small" means here, we need the following function and some numerical estimations.

Definition 3.3.15. Define the function $\rho \colon \mathbb{N} \to \mathbb{R}_{\geq 0}$ by

$$\rho(n) = \begin{cases} n & \text{if } n \in \{0, 1, 2\}\,, \\ \lfloor 2 \log_2(n-1) \rfloor & \text{if } n \geq 3\,. \end{cases}$$

Observation 3.3.16. The function $n \mapsto \rho(n)$ from Definition 3.3.15 is monotonely increasing in n.

Lemma 3.3.17. *For $n \geq 3$, the following inequalities are fulfilled:*

$$\log_2 n - 1.9 \leq \log_2(n-1)$$
$$\lfloor 2 \log_2 n \rfloor - 1 \leq \lfloor 2 \log_2(n-1) \rfloor$$
$$\lfloor 2 \log_2(n+1) \rfloor - 2 \leq \lfloor 2 \log_2(n-1) \rfloor$$

Proof. For $n = 3$, we have $\log_2 n - 1.9 < 0 < 1 = \log_2(n-1)$; and the left- and right-hand side of the second and third inequality equals 2.

For $n \geq 4$, we have

$$(n-1)\sqrt{2} - n = n\left(\sqrt{2} - 1\right) - \sqrt{2} \overset{n \geq 4}{>} 0$$

and thus $\log_2 n < \log_2(n-1) + 0.5$, which implies the first statement. Furthermore, we obtain $2 \log_2 n < 2 \log_2(n-1) + 1$. After rounding, this implies the second statement, and applied twice, it yields $2 \log_2(n+1) < 2 \log_2(n-1) + 2$, from which the third statement follows by rounding. $\square$

Proposition 3.3.18. *Let $r \in \mathbb{N}$ and a leftist circuit S on r inputs $x_0, \dots, x_{r-1}$ be given. Consider a triangular subset N of the inputs with cardinality $|N| = n \geq 1$. Let $d \in \mathbb{N}_{>0}$ be the unique integer with $2^{d-1} \leq n < 2^d$. Let $k := 2^{d-1}$. Let $K \subseteq N$ be the output of Algorithm 3.4 given S and N, and let $B := B(K, S)$ be the boundary vertices of K with respect to S. Then, the following statements are fulfilled:*

(i) We have $n - k \leq 2^{d-|B|+1} - 2$.

(ii) If $n - k \geq 1$, we have $|B| + 2\log_2(n-k) \leq 2\log_2 n$.

(iii) If $n - k \geq 2$, we have $|B| + \lfloor 2\log_2(n-k-1) \rfloor \leq \lfloor 2\log_2(n-1) \rfloor$.

(iv) For $n \geq 16$, we have $|B| + \rho(n-k) \leq \rho(n)$.

Proof. We use the notation from Algorithm 3.4.

If K is constructed in line 3, we have $|B| = 1$. This implies

$$n - k \overset{n<2^d, k\geq 1}{\leq} 2^d - 2 \overset{|B|=1}{=} 2^{d-|B|+1} - 2\,,$$

hence the first statement.

Otherwise, K is constructed in line 7. By the choice of D in line 5 and the proof of Proposition 3.3.13, in this case, K contains exactly one tree for each depth $i \in \{D+1, \dots, d-2\}$ and exactly 2 trees of depth D. This implies $|B| = (d-2) - D + 1 + 1 = d - D$. Hence, we have

$$n - k \leq 2\sum_{i=0}^{D-1} 2^i = 2\Big(2^D - 1\Big) = 2\Big(2^{d-|B|} - 1\Big) = 2^{d-|B|+1} - 2\,.$$

This shows the first statement.

Now assume that $n - k \geq 1$. Hence, there is some $\lambda \in (1, 2)$ with $n = \lambda 2^{d-1}$. From the first statement and $k = 2^{d-1}$, we obtain

$$n = k + n - k \leq 2^{d-1} + 2^{d-|B|+1} - 2\,. \tag{3.21}$$

This implies

$$\lambda = \frac{n}{2^{d-1}} \overset{(3.21)}{\leq} \frac{2^{d-1} + 2^{d-|B|+1} - 2}{2^{d-1}} \leq 1 + 2^{2-|B|}\,. \tag{3.22}$$

In order to prove the second statement, we will show that

$$2\log_2 n - 2\log_2(n-k) - |B| \geq 0\,. \tag{3.23}$$

We have

$$\begin{aligned} & 2\log_2 n - 2\log_2(n-k) - |B| \\ \overset{\text{Def. } \lambda, k}{=} \;& 2\log_2(\lambda 2^{d-1}) - 2\log_2((\lambda - 1)2^{d-1}) - |B| \\ = \;& 2\log_2 \lambda + 2(d-1) - 2\log_2(\lambda - 1) - 2(d-1) - |B| \\ = \;& 2\big(\log_2 \lambda - \log_2(\lambda - 1)\big) - |B|\,. \end{aligned} \tag{3.24}$$

The function $x \mapsto \log_2 x - \log_2(x-1)$ is strictly monotonely decreasing for $x > 1$ as its derivative $\frac{1}{\ln(2)x} - \frac{1}{\ln(2)(x-1)} = -\frac{1}{\ln(2)x(x-1)}$ is negative for all $x > 1$. Hence, we obtain

$$\begin{aligned}
& 2\log_2 n - 2\log_2(n-k) - |B| \\
\overset{(3.24)}{=} \quad & 2\big(\log_2 \lambda - \log_2(\lambda - 1)\big) - |B| \\
\overset{(3.22)}{\geq} \quad & 2\Bigg(\log_2\Big(1 + 2^{2-|B|}\Big) - \log_2\Big(1 + 2^{2-|B|} - 1\Big)\Bigg) - |B| \\
= \quad & 2\Bigg(\log_2\Big(1 + 2^{2-|B|}\Big) - 2 + |B|\Bigg) - |B| \\
= \quad & 2\log_2\Big(1 + 2^{2-|B|}\Big) + |B| - 4\,.
\end{aligned}$$

The first summand is always positive. Thus, for $|B| \geq 4$, Equation (*3.23*) follows immediately. In the other cases, we prove the statement explicitly:

$$\begin{aligned}
2\log_2\Big(1 + 2^{2-1}\Big) + 1 - 4 &= 2\log_2 3 - 3 && > 0 \\
2\log_2\Big(1 + 2^{2-2}\Big) + 2 - 4 &= 2 - 2 && = 0 \\
2\log_2\Big(1 + 2^{2-3}\Big) + 3 - 4 &= 2\log_2 1.5 - 1 && > 0
\end{aligned}$$

Hence, we have proven the second statement.

In order to see the third statement, assume that $n - k \geq 2$. Recall the above remark that the function $x \mapsto \log_2 x - \log_2(x-1)$ is strictly monotonely decreasing for $x > 1$. Together with the second statement, this implies

$$\begin{aligned}
& 2\log_2(n-1) - 2\log_2(n-k-1) - |B| \\
\geq \quad & 2\log_2(n-1) - 2\log_2 n + 2\log_2(n-k) - 2\log_2(n-k-1) \\
\geq \quad & 0\,.
\end{aligned}$$

From this, as B and 0 are integral, we obtain

$$\big\lfloor 2\log_2(n-1)\big\rfloor - \big\lfloor 2\log_2(n-k-1)\big\rfloor - |B| \geq 0\,,$$

hence the third statement.

For proving the fourth statement, we distinguish two cases.

Case 1: Assume that $n - k \leq 2$.

By the first statement, we have $n-k \leq 2^{d-|B|+1} - 2$. This implies $\log_2(n-k+2) \leq d - |B| + 1$ and thus

$$|B| \leq d + 1 - \log_2(n-k+2) \overset{k=2^{d-1}}{=} \log_2 k - \log_2(n-k+2) + 2\,. \tag{3.25}$$

From this, we obtain

$$\begin{aligned}
& \rho(n) - \rho(n-k) - |B| \\
\overset{\substack{n\geq 16,\\ n-k\leq 2}}{=} \quad & \big\lfloor 2\log_2(n-1)\big\rfloor - (n-k) - |B| \\
\overset{(3.25)}{\geq} \quad & \big\lfloor 2\log_2(n-1)\big\rfloor - (n-k) - \big(\log_2 k - \log_2(n-k+2) + 2\big) \\
\overset{n\geq 16}{\geq} \quad & \log_2(n-1) + \log_2(15) - 1 - (n-k) - \log_2 k + \log_2(n-k+2) - 2 \\
> \quad & \log_2(n-1) - (n-k) - \log_2 k + \log_2(n-k+2) + 0.9\,.
\end{aligned} \tag{3.26}$$

For $k = n$, this reduces to $\log_2(n-1) - \log_2 n + 1.9 \geq 0$, which is fulfilled by $n \geq 16$ and Lemma 3.3.17, so the fourth statement is proven. For $1 \leq n-k \leq 2$ and thus $n-2 \leq k \leq n-1$, we prove it by

$$\begin{aligned}
& \lfloor 2\log_2(n-1) \rfloor - \rho(n-k) - |B| \\
\overset{(3.26)}{\geq} \quad & \log_2(n-1) - (n-k) - \log_2 k + \log_2(n-k+2) + 0.9 \\
\overset{n-2\leq k\leq n-1}{\geq} \quad & \log_2(n-1) - 2 - \log_2(n-1) + \log_2(3) + 0.9 \\
> \quad & 0.48 \\
> \quad & 0\,.
\end{aligned}$$

Case 2: Assume that $n-k \geq 3$.

Here, by definition of ρ (see Definition 3.3.15), we have $\rho(n) = \lfloor 2\log_2(n-1) \rfloor$ since $n \geq 16 > 3$ and $\rho(n-k) = \lfloor 2\log_2(n-k-1) \rfloor$ as $n-k \geq 3$. Hence, in this case, the fourth statement coincides with the third statement.

This finishes the proof of the fourth statement and hence of this proposition. □

3.4 Size and Fanout Analysis

We shall now analyze Algorithm 3.1 (page 74) with respect to other metrics presented in Section 2.3.1. The description of Algorithm 3.1 suggests to construct each occurring symmetric circuit on r inputs with $r-1$ gates, independently from other symmetric circuits which may compute the same sub-function at intermediate stages. This yields a formula circuit for $f(s,t)$, and its properties are examined in Theorem 3.4.1. In the rest of this section, we will give an alternative implementation with a significantly lower size.

Theorem 3.4.1. *Assume that in Algorithm 3.1 (page 74), all symmetric circuits are constructed as formula circuits on the inputs. Given input variables $s = (s_0, \dots, s_{n-1})$ and $t = (t_0, \dots, t_{m-1})$ with $m \geq 2$, the circuit $\mathrm{C}(s,t)$ computed by Algorithm 3.1 is a formula circuit with*

$$\mathrm{size}(\mathrm{C}(s,t)) \leq m\,\mathrm{depth}(\mathrm{C}(s,t)) + n - 1$$

and

$$\mathrm{fanout}(\mathrm{C}(s,t)) \leq \mathrm{depth}(\mathrm{C}(s,t))\,.$$

Proof. For $m = 2$, both bounds are fulfilled since we construct a symmetric tree in line 2 with size at most $m+n-1$ and maximum fanout 1. Thus, assume that $m \geq 3$ and let $\mathcal{I}$ denote the inputs of $\mathrm{C}(s,t)$. Since $\mathrm{C}(s,t)$ is a formula circuit, Observation 2.3.6 yields $\mathrm{size}(\mathrm{C}(s,t)) = \sum_{v\in\mathcal{I}} \mathrm{fanout}(v) - 1$. Hence, both the size and fanout bound are implied by the following claim:

Claim. Assume that $m \geq 3$ and choose $d := d_{\min}(n,m) \in \mathbb{N}$ as computed in line 3 of Algorithm 3.1. In the circuit $\mathrm{C}(s,t)$, each of the symmetric inputs has fanout exactly 1 and each of the alternating inputs has fanout at most d.

Proof of claim: We prove the claim by induction on d. Note that for $d = 3$, each computed realization has maximum fanout 1.

Thus, we may assume that $d > 3$. Both in lines 13 and 17, the returned realization consists of the disjunction of two circuits on disjoint input sets. Due to the fact that any symmetric circuit has maximum fanout 1 and by induction hypothesis, in these two cases, $\mathrm{C}(s,t)$ fulfills the claimed fanout bounds.

In line 20, we return $\mathrm{C}(s,t) = \mathrm{C}\big(s,t'\big) \wedge \Big(\mathrm{C}\Big(\widehat{t'}, t''\Big)\Big)^*$. By induction hypothesis applied to the two sub-circuits, every input in s has fanout 1 in $\mathrm{C}(s,t)$, every input in t' has fanout at most $(d-1)+1 = d$ in $\mathrm{C}(s,t)$ and every input in t'' has fanout at most $d-1 < d$ in $\mathrm{C}(s,t)$.

This proves the induction step and hence the claim. □

□

Hence, the formula circuit $\mathrm{C}(s,t)$ computed by Algorithm 3.1 has a size in the order of $\mathcal{O}(m\log_2(m+n)+n)$. By sharing gates for the construction of symmetric trees using Theorem 3.3.12, we will be able to reduce this to a linear size of $\mathcal{O}(m+n)$ in Theorem 3.4.19. In order to state the algorithm, we introduce a notation for the input variables in the outermost call of Algorithm 3.1.

Definition 3.4.2. Consider the application of Algorithm 3.1 to symmetric inputs $s = \big(s_0,\dots,s_{q-1}\big)$ and alternating inputs $t = (t_0,\dots,t_{r-1})$. We define the **global inputs** $x = \big(x_0, x_2, \dots, x_{2q-2}, x_{2q}, x_{2q+1}, \dots, x_{2q+r-1}\big)$ by

$$x_i = \begin{cases} s_{\frac{i}{2}} & \text{for } i < 2q \text{ even,} \\ \text{undefined} & \text{for } i < 2q \text{ odd,} \\ t_{i-2q} & \text{for } i \geq 2q\,. \end{cases}$$

We can now use two interchangeable ways to denote the input of Algorithm 3.1: we may either apply the algorithm to symmetric inputs $s = (s_0, s_1, \dots, s_{q-1})$ and alternating inputs $t = (t_0,\dots,t_{r-1})$, or to global inputs $x = \big(x_0, x_2, \dots, x_{2q-2}, x_{2q}, x_{2q+1}, \dots, x_{2q+r-1}\big)$, and can easily convert one notation into the other. When applying Algorithm 3.1 recursively, we use the global notation for the outermost call of the algorithm, and write s and t for the inputs considered in the current recursion step. Then, this notation allows us to identify the position of the currently considered inputs s_i and t_i among the global inputs x.

Definition 3.4.3. Assume that Algorithm 3.1 (page 74) is applied to global inputs $x = \big(x_0, x_2, \dots, x_{2q-2}, x_{2q}, x_{2q+1}, \dots, x_{2q+r-1}\big)$. To each input, we assign a **parity**: We call an input x_i with $i \in \{0,\dots,2q+r-1\}$ **even** if i is even and **odd** otherwise.

Using this notation, Algorithm 3.5 states the precise algorithm: We construct large leftist And and Or circuits on the even and odd inputs, respectively (see Definition 3.3.2), and use these to construct symmetric trees during Algorithm 3.1 via Theorem 3.3.12. For this, we use that in Lemma 3.4.6, we will show that s and $s +\!\!+ (t_0)$ are both triangular. Furthermore, for the construction of the subset s' of s in line 11 of Algorithm 3.1, we use Algorithm 3.4. In Theorem 3.4.19, we shall see that this leads to a linear number of gates. The size analysis requires a deeper understanding of Algorithm 3.1. We start with an easy observation that can be verified by induction on the algorithm.

Observation 3.4.4. Assume that Algorithm 3.1 (page 74) is applied to global inputs $x = \big(x_0, x_2, \dots, x_{2q-2}, x_{2q}, x_{2q+1}, \dots, x_{2q+r-1}\big)$. In each recursive call of Algorithm 3.1 (page 74) for symmetric inputs $s = (s_0,\dots,s_{n-1})$ and alternating inputs $t = (t_0,\dots,t_{m-1})$, there is some $j \in \{0,\dots,r-1\}$ with $t_i = x_{2q+j+i}$ for all $0 \leq i \leq m-1$.

Algorithm 3.5: Depth optimization for extended And-Or paths via leftist circuits

```
Input: Global inputs $x = \big(x_0, x_2, \dots, x_{2q-2}, x_{2q}, x_{2q+1}, \dots, x_{2q+r-1}\big)$.
Output: A circuit computing $f\big((x_0, x_2, \dots, x_{2q-2}), (x_{2q}, \dots, x_{2q+r-1})\big)$.
1 Construct a leftist And-circuit $S_0$ on all even inputs $x_0, x_2, \dots$.
2 Construct a leftist Or-circuit $S_1$ on all odd inputs $x_{2q+1}, x_{2q+3}, \dots$.
3 Precompute the data from Lemma 3.3.11 for both $S_0$ and $S_1$.
4 Apply Algorithm 3.1 (page 74) to compute a circuit for
   $f\big((x_0, x_2, \dots, x_{2q-2}), (x_{2q}, \dots, x_{2q+r-1})\big)$ while constructing all arising
   symmetric circuits using Theorem 3.3.12 and computing $s'$ in line 11 via
   Algorithm 3.4.
```

In other words, t is a consecutive subset of x.

Note that Algorithm 3.1 computes a circuit for the And-Or path $f(s,t)$. In order to compute a circuit for its dual $f^*(s,t)$, we can simply call the algorithm to compute a circuit C for $f(s,t)$ and return C^*. In particular, this is what happens in line 20. As we want to use gates of the leftist circuits S_0 and S_1 in symmetric trees built during Algorithm 3.1, we need to determine the parity of the inputs in s and t depending on whether $f(s,t)$ or $f^*(s,t)$ is computed.

Lemma 3.4.5. *Assume that Algorithm 3.1 (page 74) is applied to global inputs $x = \big(x_0, x_2, \dots, x_{2q-2}, x_{2q}, x_{2q+1}, \dots, x_{2q+r-1}\big)$. Consider a recursive call of Algorithm 3.1 with symmetric inputs $s = (s_0, \dots, s_{n-1})$ and alternating inputs $t = (t_0, \dots, t_{m-1})$. Then, for the computation of $f(s,t)$ (or $f^*(s,t)$, respectively), every input in s as well as t_0 is even (or odd, respectively).*

Proof. In the outermost call of the algorithm with $s = \big(x_0, x_2, \dots, x_{2q-2}\big)$ and $t = \big(x_{2q}, x_{2q+1}, \dots, x_{2q+r-1}\big)$, the statement is true by Definition 3.4.3.

Thus assume that the statement holds in some call of the algorithm with symmetric inputs $s = (s_0, \dots, s_{n-1})$ and alternating inputs $t = (t_0, \dots, t_{m-1})$. We need to show that all recursive calls (i.e., lines 13, 17 and 20) maintain this property. It is easy to see that it suffices to show the statement for the computation of $f(s,t)$.

In this case, we can inductively assume that each input in s as well as t_0 is even. In the recursive calls computing $f\big(s \backslash s', t\big)$ (line 13) and $f((), t)$ (line 17), this immediately yields that all symmetric inputs and t_0 are even. In line 20, there are two recursive calls. For the computation of $f\big(s, t'\big)$, the symmetric inputs are again even by induction hypothesis, and so is $t'_0 = t_0$. Note that $\widehat{t'}$ consists of the inputs $t_1, t_3, t_5, \dots, t_{k-2}$ of t with k odd, and that $t''_0 = t_k$. For the recursive computation of $f^*\Big(\widehat{t'}, t''\Big)$, Observation 3.4.4 thus implies that all inputs in $\widehat{t'}$ and t''_0 are odd. □

As in line 11, we apply Algorithm 3.4 to s, we need to show that s is triangular. Furthermore, we will use in Lemma 3.4.12 that $s \mathbin{+\!\!+} (t_0)$ is triangular.

Lemma 3.4.6. *Assume that Algorithm 3.1 (page 74) is applied to global inputs $x = \big(x_0, x_2, \dots, x_{2q-2}, x_{2q}, x_{2q+1}, \dots, x_{2q+r-1}\big)$. Let a leftist* And *circuit S_0 on $x_0, x_2, \dots$ and a leftist* Or *circuit S_1 on $x_{2q+1, 2q+3}, \dots$ be given. Consider a recursive call of Algorithm 3.1 with symmetric inputs $s = (s_0, \dots, s_{n-1})$ and alternating inputs $t = (t_0, \dots, t_{m-1})$ for the computation of $f(s,t)$ (or $f^*(s,t)$, respectively). Then, the sets s and $s \mathbin{+\!\!+} (t_0)$ are triangular with respect to S_0 (or S_1, respectively).*

Proof. By Lemma 3.4.5, for the computation of $f(s,t)$ (or $f^*(s,t)$), all inputs in s and t_0 are even (or odd). Hence, they are inputs of the leftist circuit S_0 (or S_1).

By induction on the algorithm, we will prove that s and $s +\!\!+ (t_0)$ are triangular. In the outermost call, s and $s +\!\!+ (t_0)$ are both consecutive input sets of the leftist tree S_0. Hence, by Lemma 3.3.6, s and $s +\!\!+ (t_0)$ are triangular with respect to S_0.

For the inductive step, we assume without loss of generality that $f(s,t)$ is computed. Now, we may assume that s and $s +\!\!+ (t_0)$ are triangular with respect to S_0. We show that the statement remains true for each recursive call (i.e., lines 13, 17 and 20).

In line 13, we recursively realize $f(s\backslash s', t)$, where s' is computed using Algorithm 3.4. Hence, by Proposition 3.3.13, $s\backslash s'$ is triangular, and by Corollary 3.3.14, $(s\backslash s') +\!\!+ t_0$ is triangular.

In line 17, we recursively compute $f((), t)$, and the empty set and $\{t_0\}$ are both triangular by Lemma 3.3.6.

In line 20, we recursively compute $f(s,t')$ and $f^*(\widehat{t'}, t'')$. For $f(s,t')$, the statement is true by induction hypothesis. By Observation 3.4.4, t is a consecutive subset of the global inputs, and by Lemma 3.4.5, t_0 is even. Thus, the inputs of $\widehat{t'}$ and t''_0 are all odd and a consecutive subset of inputs of S_1. Hence, by Lemma 3.3.6, both $\widehat{t'}$ and $\widehat{t'} +\!\!+ (t''_0)$ are triangular with respect to S_1.

This proves the induction step and hence the lemma. □

In order to prove that the size of our AND-OR path circuits is linear in the number of inputs, we partition the gates into groups and estimate how many gates are used per group.

Definition 3.4.7. Consider the circuit $\mathrm{C}(s,t)$ computed by Algorithm 3.5 for symmetric input s and alternating inputs t. We distinguish five types of gates used in $\mathrm{C}(s,t)$:

(i) gates of the leftist circuits S_0 and S_1 in lines 1 and 2 of Algorithm 3.5

(ii) one concatenation gate per any alternating split in line 20 of Algorithm 3.1

(iii) gates constructed in base-case solutions in lines 1 to 8 of Algorithm 3.1

(iv) gates used in symmetric circuits in line 13 or line 17 of Algorithm 3.1

(v) one concatenation gate per any split in line 13 or line 17 of Algorithm 3.1

We also call the gates of types (iii) to (v) **additional gates**.

Note that this indeed defines a partition of all the gates used in $\mathrm{C}(s,t)$. Counting the gates of types (i) and (ii) will be easy, the important step will be counting the additional gates of types (iii) to (v).

Lemma 3.4.8. *Consider the circuit* $\mathrm{C}(s,t)$ *computed by Algorithm 3.5 for symmetric input s and alternating inputs t. For $m \geq 2$, the circuit* $\mathrm{C}(s,t)$ *contains at most $m+n-2$ gates of type (i).*

Proof. The gates of type (i) are the gates contained in the two symmetric circuits S_0 and S_1 computed in lines 1 to 2 of Algorithm 3.5. Note that as $m \geq 2$, the circuit S_0 has exactly $n + \lceil \frac{m}{2} \rceil$ inputs and the circuit S_1 has exactly $\lfloor \frac{m}{2} \rfloor$ inputs. For both

S_0 and S_1, the number of gates is at most the number of inputs minus 1. Hence, the total number of gates in S_0 and S_1 is most

$$n + \left\lceil \frac{m}{2} \right\rceil - 1 + \left\lfloor \frac{m}{2} \right\rfloor - 1 = n + m - 2\,. \qquad \square$$

In order to count the gates of type (ii), we prove that there are only linearly many alternating splits in Algorithm 3.1.

Observation 3.4.9. By induction, one can see that when Algorithm 3.1 (page 74) is called for alternating inputs $r \geq 1$, then any m considered during recursive calls fulfills $m \geq 1$.

Lemma 3.4.10. *Assume that Algorithm 3.1 (page 74) is applied to global inputs $x = \big(x_0, x_2, \dots, x_{2q-2}, x_{2q}, x_{2q+1}, \dots, x_{2q+r-1}\big)$ with $r \geq 1$. Then, the number of alternating splits used in Algorithm 3.1 is at most $r - 1$.*

Proof. We prove the statement by induction on the execution of Algorithm 3.1. Consider a call of Algorithm 3.1 to inputs $s = (s_0, \dots, s_{n-1})$ and $t = (t_0, \dots, t_{m-1})$.

In all the base cases (i.e., $m \leq 2$ or $d \leq 3$), we do not perform an alternating split, thus the statement is valid since we always have $m \geq 1$ by Observation 3.4.9.

Thus, assume that we compute $C(s,t)$ recursively. If we use any of the splits in line 13 or line 17, we do not perform an alternating split in the current recursive call. As in both splits, we apply recursion to some set of symmetric inputs and alternating inputs t, by induction hypothesis, we perform at most $m-1$ alternating splits during the computation of $f(s,t)$. Finally, consider the case that we use an alternating split

$$C(s,t) = C\big(s,t'\big) \wedge \left(C\left(\widehat{t'}, t''\right)\right)^*$$

with $t' = (t_0, \dots, t_{k-1})$ in line 20. By induction hypothesis, we perform at most $k-1$ alternating splits for the computation of $C\big(s,t'\big)$, and at most $m-k-1$ alternating splits for the computation of $\left(C\left(\widehat{t'}, t''\right)\right)^*$. Adding the current alternating split, we need at most $m-1$ alternating splits for the computation of $C(s,t)$. $\square$

From this, the number of gates of type (ii) directly follows.

Corollary 3.4.11. *Consider the circuit $\mathrm{C}(s,t)$ computed by Algorithm 3.5 (page 89) for symmetric input s and alternating inputs t. For $m \geq 1$, the circuit $\mathrm{C}(s,t)$ contains at most $m-1$ gates of type (ii).* $\square$

In the following lemma, we estimate the number of additional gates for small values of n and m. Here, we need to examine the concrete realizations constructed for $f(s,t)$.

Lemma 3.4.12. *Consider the circuit $\mathrm{C}(s,t)$ computed by Algorithm 3.5 (page 89) for symmetric inputs $s = (s_0, \dots, s_{n-1})$ and alternating inputs $t = (t_0, \dots, t_{m-1})$ with $m \geq 1$. The number of additional gates (types (iii) to (v)) needed for the construction of $\mathrm{C}(s,t)$ is shown in Table 3.2 for the following values of m and n:*

- *$1 \leq m \leq 2$ and $n \in \mathbb{N}$ arbitrary*
- *$d \leq 4$ and all $m, n \in \mathbb{N}$ with $d_{\min}(n,m) = d$*

m \ n	0	1	2	3	4	5	6	7	8	9	10	n > 10
1	🟦	🟦	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥
2	🟦	🟦	🟥	🟥	🟥	🟥	🟥	🟥 $m + 2\log_2(n+1) - 3$	🟥	🟥	🟥	🟥
3	🟦	🟦	🟥	🟥	🟨	🟨	🟨	🟨	🟨	🟩 $m + 5$	🟩	
4	🟦	🟦 $m + n - 1$	🟦	🟨	🟨	🟨 $m + 2\log_2 n - 2$	🟨					
5	🟦	🟦	🟨									
6	🟦	🟦	🟨	🟨	🟨	🟨	🟨	🟨	🟨	🟩	🟩	
7	🟦	🟦	🟨	🟨	🟨	🟨	🟨					
8	🟦											

Table 3.2: Number of additional gates needed for the construction of $\mathrm{C}(s, t)$ in the cases $3 \leq m \leq 8$ and $0 \leq n \leq 10$ considered in Lemma 3.4.12. Cells with the same color contain the same formula for the number of additional gates.

- *$d = 5$, $6 \leq m \leq 8$ and all $n \in \mathbb{N}$ with $d_{\min}(n, m) = d$*

Proof. For the values $d_{\min}(n, m)$, see Table 3.1 (page 64).

Case 1: Assume that $1 \leq m \leq 2$.

In this case, we compute $\mathrm{C}(s, t)$ as an optimum symmetric tree on $m + n \geq 1$ inputs in line 2 of Algorithm 3.1 (page 74). By Lemma 3.4.6, the set $s +\!\!\!+ (t_0)$ is triangular with respect to S_0. Setting $K = s +\!\!\!+ (t_0)$ and $L = (t_1, \dots, t_{m-1})$, we have $|K| = n + 1 \geq 3$ and $|L| = m - 1$. By Theorem 3.3.12, for the construction of $\mathrm{C}(s, t)$, we need at most $m + n - 1$ gates if $n \leq 1$, and $2\log_2(n+1) + m - 1 - 2 = m + 2\log_2(n+1) - 3$ additional gates otherwise.

This bounds the number of additional gates for $m \in \{1, 2\}$, i.e., the first two rows of Table 3.2. In particular, by Table 3.1, this covers all cases of m and n with $d_{\min}(n, m) = 1$.

Case 2: Assume that $m \geq 3$.

Let $d := d_{\min}(n, m)$ as in line 3 of Algorithm 3.1. We traverse the remaining cases of m and n in order of increasing d. Thus, we may assume that the number of additional gates needed in the circuit $\mathrm{C}(m', n')$ as shown in Table 3.2 with $d_{\min}(m', n') < d$ has already been verified.

Case 2.1: Assume that $d \leq 3$.

Here, we construct $\mathrm{C}(s, t)$ in line 6 or line 8 of Algorithm 3.1. If $m = 4$ and thus $n = 0$ by Table 3.1, we construct a standard circuit using $m + n - 1$ gates in line 8. Otherwise, by Table 3.1, we have $m = 3$ and $n \leq 3$. By Lemma 3.4.6, the set $s +\!\!\!+ (t_0)$ is triangular with respect to S_0. Hence, we can compute a delay-optimum symmetric circuit for $\mathrm{sym}\big((s_0, \dots, s_{n-1}, t_0, t_1 \vee t_2)\big)$ via Theorem 3.3.12 in line 6. Note that this is a depth optimum circuit for $f(s, t)$. By Theorem 3.3.12, this requires $m + n \geq 1$ gates if $n \leq 1$ and $2\log_2(n+1) + m - 1 - 2 = m + 2\log_2(n+1) - 3$ additional gates otherwise.

Case 2.2: Assume that $d = 4$.

Here, Table 3.1 yields $m \leq 5$.

Case 2.2.1: Assume that $n \geq 2^{d-1}$.

As $n \geq 2^{d-1} = 8$, Table 3.1 together with $m \geq 3$ implies $m = 3$ and $n \leq 10$, and we apply the symmetric split $f(s, t) = \mathrm{sym}(s') \wedge f(s \backslash s', t)$ in line 13 of Algorithm 3.1, where s' with $|s'| = 8$ is computed by Algorithm 3.4. By Proposition 3.3.13, both s' and $s \backslash s'$ are triangular. Using Theorem 3.3.12 to construct a circuit for $\mathrm{sym}(s')$, we need at most $2\log_2 8 - 2 = 4$ additional gates. As $n - 8 \leq 2$ and $m = 3$, for constructing $\mathrm{C}(s \backslash s', t)$, we need at most

- $m + n - 8 - 1 = m + n - 9$ additional gates if $n - 8 \in \{0, 1\}$,
- $\lfloor m + 2\log_2(3) - 3 \rfloor = 3$ additional gates if $n - 8 = 2$

by the already computed numbers of additional gates in Table 3.2 for $d \in \{2, 3\}$. Adding the split gate, in total, if $n = 8$, this makes at most $4+m+n-9+1 = m+4 = m+2\log_2 n-2$ additional gates; if $n = 9$, this makes at most $4+m+n-9+1 = m+5$ additional gates; and if $n = 10$, this makes at most $4 + 3 + 1 = m + 5$ additional gates.

Case 2.2.2: Assume that $n < 2^{d-1}$ and $m \leq \mu(4 - 1, 0)$.

Note that we have $n < 2^{d-1} = 8$ and $3 \leq m \leq \mu(4-1, 0) = \frac{8-2}{3} + 2 = 4$. We use the simple split

$$f(s, t) = \mathrm{sym}(s) \wedge f\big((), (t_0, \dots, t_{m-1})\big)$$

in line 17 of Algorithm 3.1. By Table 3.1, we have $4 \leq n \leq 7$ for $m = 3$ and $1 \leq n \leq 6$ for $m = 4$. Here, as s is triangular by Lemma 3.4.6, by Theorem 3.3.12, for realizing $\mathrm{sym}(s)$, if $n \in \{1, 2\}$, we need $n - 1$ gates, and at most $2\log_2 n - 2$ additional gates otherwise; and at most $m - 1$ gates for $\mathrm{C}\big((), (t_0, \dots, t_{m-1})\big)$ by Table 3.2. In total, if $m = 4$ and $n \in \{1, 2\}$, we need exactly $n - 1 + m - 1 + 1 = m + n - 1$ additional gates, and otherwise at most $2\log_2 n - 2 + m - 1 + 1 = m + 2\log_2 n - 2$ additional gates.

Case 2.2.3: Assume that $n < 2^{d-1}$ and $m > \mu(4 - 1, 0)$.

If $n < 2^{d-1} = 8$ and $m > \mu(d-1, 0) = 4$, we have $m = 5$ and $n \leq 2$ by Table 3.1. For any $0 \leq n \leq 2$, we choose $k = 3$ in line 18 since

$$3 < \frac{10}{3} = \frac{2^3 - 2 - 2}{3} + 2 \leq \mu(d-1, n) \leq \frac{2^3 - 2}{3} + 2 = 4\,.$$

Hence, we perform the alternating split

$$\mathrm{C}(s, t) = \mathrm{C}\big(s, (t_0, t_1, t_2)\big) \wedge \mathrm{C}^*\big((t_1), (t_3, t_4)\big)$$

in line 20 of Algorithm 3.1. Recall that in this case, the concatenation gate is no additional gate by Definition 3.4.7. For computing $\mathrm{C}\big(s, (t_0, t_1, t_2)\big)$, by Table 3.2, if $n \leq 1$, we need at most $3 + n - 1 = n + 2$ additional gates, and if $n = 2$, we need at most $\lfloor 3 + 2\log_2(3) - 3 \rfloor = 3$ additional gates. By Table 3.2, the computation of $\mathrm{C}^*\big((t_1), (t_3, t_4)\big)$ requires $2 + 1 - 1 = 2$ gates. In total, if $n \leq 1$, we need at most $n + 2 + 2 \overset{m=5}{=} m + n - 1$ additional gates for constructing $f(s, t)$, and if $n = 2$, we need at most $3 + 2 = 5 = m + 2\log_2 n - 2$ additional gates for constructing $f(s, t)$.

Case 2.3: Assume that $d = 5$.

Here, we have $6 \leq m \leq 8$ by assumption, and Table 3.1 implies

- $n \leq 10$ for $m = 6$,
- $n \leq 5$ for $m = 7$, and
- $n = 0$ for $m = 8$.

In any case, we have $n < 2^{d-1} = 16$, and we perform the alternating split in line 20 of Algorithm 3.1 since by Table 3.1, we have $d_{\min}(0, m) = d$. Hence, we choose k maximal with $k \leq \mu(d-1, n) = \frac{16-n-2}{4} + 2 = 5.5 - \frac{n}{4}$. For $0 \leq n \leq 2$, this means $k = 5$, while for $3 \leq n \leq 10$, this means $k = 3$.

Case 2.3.1: Assume that $n \leq 2$.

Here, we have $k = 5$ and perform the alternating split

$$\mathrm{C}(s,t) = \mathrm{C}\big(s,(t_0,t_1,t_2,t_3,t_4)\big) \wedge \mathrm{C}^*\big((t_1,t_3),(t_5,\ldots,t_{m-1})\big)\,.$$

We read off the number of additional gates used for the two sub-circuits from Table 3.2: For $\mathrm{C}\big(s,(t_0,t_1,t_2,t_3,t_4)\big)$, if $n \leq 1$, we need at most $5+n-1 = n+4$ additional gates, and, if $n = 2$, we need at most $5+2\log_2 n-2 = 5$ additional gates. For $\mathrm{C}^*\big((t_1,t_3),(t_5,\ldots,t_{m-1})\big)$ and hence $m-5 \in \{1,2\}$ alternating inputs, we need at most $\lfloor m-5+2\log_2(2+1)-3\rfloor = m-5$ additional gates. Thus, in total, if $n \in \{0,1\}$, we need at most $n+4+m-5 = m+n-1$ additional gates, and if $n = 2$, we need at most $5+m-5 = m+2\log_2 n-2$ additional gates.

Case 2.3.2: Assume that $3 \leq n \leq 10$.

Now, we have $k = 3$ and perform the alternating split

$$\mathrm{C}(s,t) = \mathrm{C}\big(s,(t_0,t_1,t_2)\big) \wedge \mathrm{C}^*\big((t_1),(t_3,\ldots,t_{m-1})\big)\,.$$

The circuit $\mathrm{C}\big(s,(t_0,t_1,t_2)\big)$ is constructed using the following number of additional gates as already computed in Table 3.2:

- For $n = 3$, we need $3+2\log_2(3+1)-3 = 4$ additional gates.
- For $4 \leq n \leq 8$, we need at most $3+2\log_2 n-2 = 2\log_2 n+1$ additional gates.
- For $n \in \{9,10\}$, we need at most $3+5 = 8$ additional gates.

The circuit $\mathrm{C}^*\big((t_1),(t_3,\ldots,t_{m-1})\big)$ can be built using at most $m-3+1-1 = m-3$ additional gates by Table 3.2. Summing up these numbers, we obtain the last entries of Table 3.2:

- For $m \in \{6,7\}$ and $n = 3$, we need at most $4+m-3 = m+1 = \lfloor m+2\log_2 n-2\rfloor$ additional gates.
- For $m = 6$ and $4 \leq n \leq 8$, and for $m = 7$ and $4 \leq n \leq 5$, we need at most $2\log_2 n+1+m-3 = m+2\log_2 n-2$ additional gates.
- For $m = 6$ and $n \in \{9,10\}$ we need at most $8+m-3 = m+5$ additional gates.

This bounds the number of additional gates used for the computation of $f(s,t)$ for all stated cases of m and n. □

We now give a common upper bound for the number of additional gates in the cases considered in Lemma 3.4.12 and Table 3.2 in Corollary 3.4.13.

Corollary 3.4.13. *Consider the circuit* $\mathrm{C}(s,t)$ *computed by Algorithm 3.5 (page 89) for symmetric inputs* $s = (s_0,\ldots,s_{n-1})$ *and alternating inputs* $t = (t_0,\ldots,t_{m-1})$ *with* $m \geq 1$*. For all* m,n *appearing in Table 3.2 (see Lemma 3.4.12), the number of additional gates in* $\mathrm{C}(s,t)$ *is at most* $m+\rho(n)-1$*.*

Proof. We partition all cases to consider based on the coloring of Table 3.2.

First consider the red part, i.e., the cases $1 \leq m \leq 2$ and $n \geq 2$, and $m = 3$, $n \in \{2,3\}$. Here, by Table 3.2, we need at most $m+2\log_2(n+1)-3$ additional gates. If $n \leq 2$, we have at most

$$\lfloor m+2\log_2(3)-3\rfloor = m < m+1 \leq m+\rho(n)-1$$

additional gates, and for $n \geq 3$, we have

$$\lfloor m + 2\log_2(n+1) - 3 \rfloor \overset{\substack{n\geq 3,\\ \text{Lem. 3.3.17}}}{\leq} m + \lfloor 2\log_2(n-1) \rfloor - 1 \overset{n\geq 3}{=} m + \rho(n) - 1\,.$$

Now consider the blue part, i.e., the cases $m \in \{1, \dots, 7\}$ and $n \in \{0, 1\}$, $m = 4$ and $n = 2$, or $m = 8$ and $n = 0$. This implies that $n \leq 2$, hence we need at most $m + n - 1 \overset{n\leq 2}{=} m + \rho(n) - 1$ additional gates.

For the yellow part (i.e., $m = 3$ and $n \in \{4, \dots, 8\}$; $m = 4$ and $n \in \{3, \dots, 6\}$; $m = 5$ and $n = 2$; $m = 6$ and $n \in \{2, \dots, 8\}$; $m = 7$ and $n \in \{2, \dots, 5\}$), we need at most $\lfloor m + 2\log_2 n - 2 \rfloor$ additional gates. For $n = 2$, we have $m + 2\log_2 n - 2 = m < m + 2 - 1 = m + \rho(n) - 1$. for $3 \leq n \leq 8$, we have

$$\lfloor m + 2\log_2 n - 2 \rfloor \overset{\substack{n\geq 3\\ \text{Lem. 3.3.17}}}{\leq} m + \lfloor 2\log_2(n-1) \rfloor - 1 \overset{n\geq 3}{=} m + \rho(n) - 1\,.$$

For the green part (i.e., $m \in \{3, 6\}$ and $n \in \{9, 10\}$), we need at most

$$m + 5 \overset{n\geq 9}{\leq} m + 2\log_2(n-1) - 1 \overset{n\in\{9,10\}}{=} m + \rho(n) - 1$$

additional gates.

Note that this a complete enumeration of all cases by Table 3.2. □

For general n and m, we will estimate the number of additional gates needed for the realization of $f\big((s_0, \dots, s_{n-1}), (t_0, \dots, t_{m-1})\big)$ in Lemma 3.4.18, but we still need some technical preparations. The following lemmas introduce important functions that are used when proving Lemma 3.4.18. Note that we show three of these functions in Figure 3.5.

Lemma 3.4.14. *For $n \in \mathbb{N}_{\geq 1}$, consider the finite series $S_n := \sum_{k=2}^{n} \frac{(k-1)^2}{2^{k-2}}$. We have*

$$S_n = 12 - \frac{4}{2^n}\left(n^2 + 2n + 3\right). \tag{3.27}$$

Moreover, we have

$$\sum_{k\geq 19} \frac{(k-1)^2}{2^{k-2}} \leq 0.006\,. \tag{3.28}$$

Proof. We prove Equation (3.27) by induction on n.

For $n = 2$, we have

$$S_n = 1 = 12 - \frac{4}{4}(4 + 4 + 3)\,.$$

For $n \geq 2$, we have

$$\begin{aligned}
S_{n+1} &= \sum_{k=2}^{n+1} \frac{(k-1)^2}{2^{k-2}} \\
&\overset{\text{(IH)}}{=} 12 - \frac{4}{2^n}\left(n^2 + 2n + 3\right) + \frac{n^2}{2^{n-1}} \\
&= 12 - \frac{4}{2^n}\left(0.5n^2 + 2n + 3\right) \\
&= 12 - \frac{4}{2^{n+1}}\left(n^2 + 4n + 6\right) \\
&= 12 - \frac{4}{2^{n+1}}\left((n+1)^2 + 2(n+1) + 3\right).
\end{aligned}$$

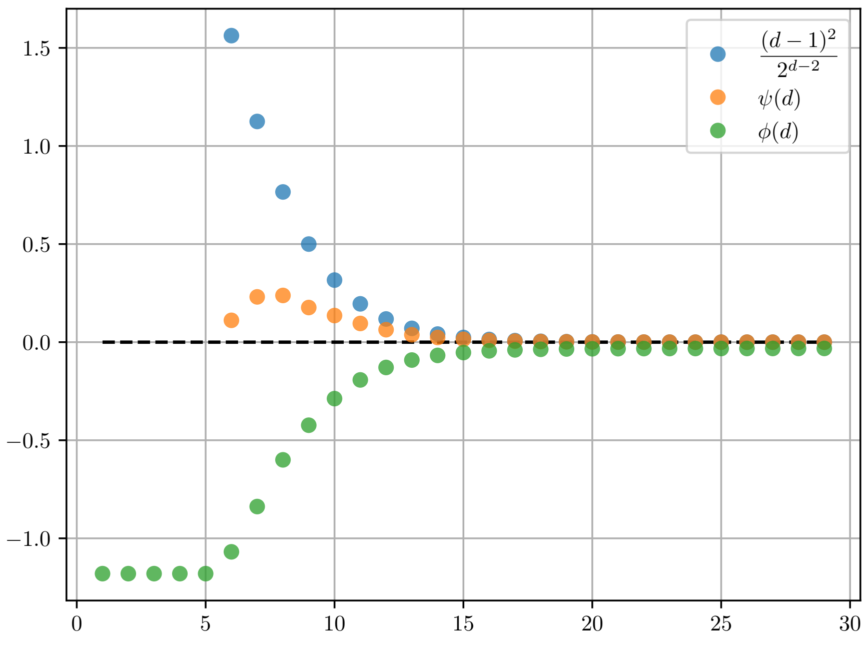


Figure 3.5: The functions $\frac{(d-1)^2}{2^{d-2}}$ for $d \geq 6$ from Lemma 3.4.14, $\psi\colon \mathbb{N}_{\geq 6} \to \mathbb{R}$ from Lemma 3.4.16, and $\phi\colon \mathbb{N}_{\geq 1} \to \mathbb{R}$ from Lemma 3.4.17.

This proves the induction step and hence Equation (3.27).

To see that Equation (3.28) is fulfilled, note that Equation (3.27) implies

$$\sum_{k=2}^{\infty} \frac{(k-1)^2}{2^{k-2}} \leq 12\,. \tag{3.29}$$

From this, we conclude

$$\begin{aligned}
\sum_{k=19}^{\infty} \frac{(k-1)^2}{2^{k-2}} &= \sum_{k=2}^{\infty} \frac{(k-1)^2}{2^{k-2}} - \sum_{k=2}^{18} \frac{(k-1)^2}{2^{k-2}} \\
&\overset{(3.29)}{\leq} 12 - \sum_{k=2}^{18} \frac{(k-1)^2}{2^{k-2}} \\
&\overset{(3.27)}{=} 12 - \left(12 - \frac{4}{2^{18}}\left(18^2 + 2 \cdot 18 + 3\right)\right) \\
&= \frac{1304}{2^{18}} \\
&< 0.006\,.
\end{aligned}$$

□

Notation 3.4.15. For $x \in \mathbb{R}$, let $\operatorname{flodd}(x) := \max\{y \in \mathbb{Z} : y \text{ odd}, y \leq x\}$.

Lemma 3.4.16. *For $d \in \mathbb{N}$, $d \geq 6$, and $\gamma \leq 0$, define $\psi(d) \in \mathbb{R}_{\geq 0}$ by*

$$\psi(d) := \frac{\rho\left(\frac{\operatorname{flodd}\left(\frac{2^{d-1}-2}{d-1}\right)+1}{2}\right) + \gamma}{\left\lfloor \frac{2^{d-1}-2}{d-1} \right\rfloor + 3}\,.$$

We have

$$\psi(d) \leq \frac{(d-1)^2}{2^{d-2}}\,.$$

Proof. We have

$$2^{d-1} \cdot 2(d-1) - (2^{d-1} + d - 3) = 2^{d-1}(d-3) + d(2^{d-1} - 1) + 3 \overset{d \geq 6}{>} 0 \tag{3.30}$$

and hence

$$\frac{\operatorname{flodd}\left(\frac{2^{d-1}-2}{d-1}\right) + 1}{2} \leq \frac{\frac{2^{d-1}-2}{d-1} + 1}{2} = \frac{2^{d-1} + d - 3}{2(d-1)} \overset{(3.30)}{\leq} 2^{d-1}\,. \tag{3.31}$$

Furthermore, we have

$$\left\lfloor \frac{2^{d-1}-2}{d-1} \right\rfloor + 3 \geq \frac{2^{d-1}-2}{d-1} + 2 = \frac{2^{d-1} + 2d - 4}{d-1} \overset{d \geq 2}{\geq} \frac{2^{d-1}}{d-1}\,. \tag{3.32}$$

As both the nominator and denominator of $\psi(d)$ are positive for $d \geq 6$, we can conclude from these inequalities that

$$\begin{aligned}
\psi(d) \quad &= \quad \frac{\rho\left(\frac{\operatorname{flodd}\left(\frac{2^{d-1}-2}{d-1}\right)+1}{2}\right) + \gamma}{\left\lfloor \frac{2^{d-1}-2}{d-1} \right\rfloor + 3} \\
&\overset{\substack{(3.31),\\ \text{Obs. } 3.3.16}}{\leq} \quad \frac{\rho\left(2^{d-1}\right) + \gamma}{\left\lfloor \frac{2^{d-1}-2}{d-1} \right\rfloor + 3} \\
&\overset{\substack{\text{Def. } 3.3.15\\ (3.32)}}{\leq} \quad \frac{\left\lfloor 2\log_2\left(2^{d-1} - 1\right) \right\rfloor - \gamma}{\frac{2^{d-1}}{d-1}} \\
&\overset{\gamma \leq 0}{\leq} \quad \frac{2(d-1)^2}{2^{d-1}}\,.
\end{aligned}$$

□

In the following lemma, we will require an upper bound on $\psi(d)$ for all $d \geq 6$. The previous lemma suggests to use $\frac{(d-1)^3}{2^{d-1}}$ as this upper bound. But in Figure 3.5, we see that the difference $\frac{(d-1)^3}{2^{d-1}} - \psi(d)$ is very large for small d, though quickly decreases to a value close to 0. Hence, in the following proof, we evaluate $\psi(d)$ explicitly for $d \leq 18$ in Table 3.3 and use the upper bound from Lemma 3.4.16 only for $d \geq 19$.

Lemma 3.4.17. *Define the function* $\phi\colon \mathbb{N}_{\geq 1} \to \mathbb{R}$ *by*

$$\phi(d) = \begin{cases} -1.15 & \text{for } d \leq 5, \\ \phi(d-1) + \psi(d) & \text{for } d \geq 6. \end{cases}$$

Then, $\phi(d)$ *is negative and monotonely increasing for all* $d \geq 1$.

Proof. Note that the correlation of $\phi(d)$ and $\psi(d)$ for $d \geq 6$ is visualized in Figure 3.5.

d	Upper bound on $\psi(d)$	Upper bound on $\sum_{d'=6}^{d} \psi(d')$
6	0.1112	0.1112
7	0.2308	0.3420
8	0.2381	0.5801
9	0.1765	0.7566
10	0.1356	0.8922
11	0.0953	0.9875
12	0.0635	1.0510
13	0.0378	1.0888
14	0.0237	1.1125
15	0.0145	1.1270
16	0.0087	1.1357
17	0.0049	1.1406
18	0.0029	1.1435

Table 3.3: Upper bounds on $\psi(d)$ and $\sum_{d'=6}^{d} \psi(d')$ for $d \in \{6, \ldots, 18\}$. All bounds on ψ have been calculated by a C++ program using floating-point interval arithmetic and rounding to fixed precision afterwards. The upper bound 1.1435 on $\sum_{d'=6}^{18} \psi(d')$ is used in the proof of Lemma 3.4.17.

As $\psi(d) \geq 0$ for all $d \geq 6$, the second statement is clearly fulfilled. Hence, in order to prove the first statement, we show that

$$\phi(5) + \sum_{d \geq 6} \psi(d) < 0\,.$$

But we have

$$\begin{aligned}
\phi(5) + \sum_{d \geq 6} \psi(d) &= \phi(5) + \sum_{d=6}^{18} \psi(d) + \sum_{d \geq 19} \psi(d) \\
&\overset{\text{Lem. 3.4.16}}{\leq} \phi(5) + \sum_{d=6}^{18} \psi(d) + \sum_{d \geq 19} \frac{(d-1)^2}{2^{d-2}} \\
&\overset{\substack{\text{Table 3.3,} \\ (3.28)}}{\leq} \phi(5) + 1.1435 + 0.006 \\
&< \phi(5) + 1.15 \\
&= 0\,.
\end{aligned}$$

□

Now, in Lemma 3.4.18, we can finally bound the number of additional gates used in the circuit $\mathrm{C}\big((s_0, \ldots, s_{n-1}), (t_0, \ldots, t_{m-1})\big)$ by $\alpha m + \rho(n) + \gamma$ for some $\alpha \geq 1$ and $\gamma \in \mathbb{R}$ to be defined. We prove the statement by induction on $d := d_{\min}(n, m)$, and in order to make the induction step work, we need to show a stronger upper bound on the number of additional gates, namely $\big(\alpha + \phi(d)\big)m + \rho(n) + \gamma$. This is a stronger bound since $\phi(d) \leq 0$ by Lemma 3.4.17, and we will see in the following proof that $\phi(d)$ is defined in a way such that

- $\alpha + \phi(d) = 1$ for $d \leq 5$, which ensures that the upper bound is valid in the case of small d (see also Corollary 3.4.13), i.e., cases 1 and 2.2.2.1 of the proof,
- and $\phi(d) = \phi(d-1) + \psi(d)$ for $d \geq 6$, which will be the definition needed for $d \geq 6$ in case of the alternating split in line 20 of Algorithm 3.1, i.e., in case 2.2.2.2 of the proof.

Lemma 3.4.18. *Given input variables $s = (s_0, \dots, s_{n-1})$ and $t = (t_0, \dots, t_{m-1})$ with $m \geq 1$, the number of additional gates (types (iii) to (v) of Definition 3.4.7) used in $\mathrm{C}(s,t)$ is at most*

$$\Phi(d,m,n) := \big(\alpha + \phi(d)\big)m + \rho(n) + \gamma\,,$$

where $d := d_{\min}(n,m)$ and $\phi(d)$ as in Lemma 3.4.17, $\rho(n)$ as in Definition 3.3.15, $\alpha = 2.15$, and $\gamma = -1$.

Proof. Figure 3.5 (page 96) depicts the function $\phi(d)$ from Lemma 3.4.17.

Case 1: Assume that $d \leq 4$ or $m \leq 2$.

By Corollary 3.4.13, we need at most $m + \rho(n) - 1$ additional gates in this case. By Lemma 3.4.17, we have $\phi(d) \geq -1.15 = -\alpha + 1$ for all $d \in \mathbb{N}_{>1}$. Hence, we have

$$\Phi(d,m,n) = \big(\alpha + \phi(d)\big)m + \rho(n) + \gamma \geq m + \rho(n) - 1\,,$$

which finishes the proof when $d \leq 4$ or $m \leq 2$.

Case 2: Assume that $m \geq 3$ and $d \geq 5$.

We follow the course of Algorithm 3.1 (page 74). Note that the value d in our proof coincides with the value d chosen in Algorithm 3.1, and recall that by Lemma 3.1.11, we have $n < 2^d$ since $m \geq 2$.

Case 2.1: Assume that $n \geq 2^{d-1}$.

The assumption $d \geq 5$ implies that $n \geq 2^{d-1} \geq 16$. In this case, we perform a symmetric split in line 13. Recall that $k = 2^{d-1}$. By Lemma 3.4.5, all symmetric inputs are even, and by Lemma 3.4.6, s is triangular. Let s' be the output of Algorithm 3.4 when applied to input set s' and leftist tree S_0 as in line 11 of Algorithm 3.1. We construct the symmetric tree on s' using Proposition 3.3.10 with $|B| - 1$ additional gates, where $B := B(s', S)$ is the set of boundary vertices of s' with respect to S. By induction hypothesis, we can assume that we need at most $\Phi(d-1, m, n-k)$ additional gates for the computation of $f\big((s_k, \dots, s_{n-1}), t\big)$. Adding the concatenation gate, we in total use $|B| + \Phi(d-1,m,n-k)$ additional gates. Note that

$$\begin{aligned}
& \Phi(d,m,n) - \big(|B| + \Phi(d-1,m,n-k)\big) \\
= \quad & \big(\alpha + \phi(d)\big)m + \rho(n) + \gamma - |B| - \big(\alpha + \phi(d-1)\big)m - \rho(n-k) - \gamma \\
\overset{\text{Lem. 3.4.17}}{\geq} \quad & \rho(n) - \rho(n-k) - |B| \\
\overset{\substack{n \geq 16,\\ \text{Prop. 3.3.18,}(iv)}}{\geq} \quad & 0\,.
\end{aligned}$$

Hence, the number of additional gates used in this case is at most $\Phi(d,m,n)$.

Case 2.2: Assume that $n < 2^{d-1}$.

Case 2.2.1: Assume that $m \leq \mu(d-1, 0)$.

Note that $n > 0$ as $m > \mu(d-1,0)$ by the choice of $d = d_{\min}(m,n)$. We construct a realization in line 17 using the simple split

$$f(s,t) = \operatorname{sym}\big((s_0,\dots,s_{n-1})\big) \wedge f\big((),t\big)\,.$$

By induction hypothesis, the circuit for $f\big((),t\big)$ requires $\Phi(d-1,m,0)$ additional gates. Denoting the number of additional gates needed in the circuit for $f\big((s_0,\dots,s_{n-1})\big)$ by σ, we need at most $\sigma + \Phi(d-1,m,0)+1$ additional gates in this case. We need to show that this is at most $\Phi(d,m,n)$. As we have

$$\begin{aligned} & \Phi(d,m,n) - \big(\sigma + \Phi(d-1,m,0)+1\big) \\ = \;& (\alpha+\phi(d))m + \rho(n) + \gamma - \big(\sigma + (\alpha+\phi(d-1))m + \rho(0) + \gamma + 1\big) \\ \overset{\rho(0)=0}{=} \;& (\phi(d)-\phi(d-1))m + \rho(n) - \sigma - 1 \\ \overset{\text{Lem. 3.4.17}}{\geq} \;& \rho(n) - \sigma - 1\,, \end{aligned}$$

it remains to show that $\rho(n) - \sigma - 1 \geq 0$.

As s is triangular by Lemma 3.4.6, we can construct the symmetric tree on s via Theorem 3.3.12 using at most $n-1$ gates if $n \leq 2$ and at most $\sigma \leq \lfloor 2\log_2 n \rfloor - 2$ additional gates otherwise. For $n \leq 2$, this shows the statement as $\rho(n) = n$ in this case. For $n \geq 3$, as $\rho(n) = \big\lfloor 2\log_2(n-1)\big\rfloor$, we have

$$\rho(n) - \sigma - 1 \geq \big\lfloor 2\log_2(n-1)\big\rfloor - \lfloor 2\log_2 n\rfloor + 2 - 1 \overset{\substack{n\geq 3,\\ \text{Lem. 3.3.17}}}{\geq} 0\,.$$

Case 2.2.2: Assume that

$$m > \mu(d-1,0)\,, \tag{3.33}$$

i.e., that we perform an alternating split

$$f(s,t) = f\big(s,t'\big) \wedge \Big(f\Big(\widehat{t'},t''\Big)\Big)^*$$

with a prefix of odd length k in line 20. We consider the case $d=5$ separately.

Case 2.2.2.1: Assume that $d=5$.

Here, Table 3.1 yields $m \leq 8$. Furthermore, Equation (3.33) implies

$$m > \mu(d-1,0) = \frac{2^4-2}{4} + 2 = 5.5$$

and thus $m \in \{6,7,8\}$. Hence, Tables 3.1 and 3.2 contain all relevant cases. By Corollary 3.4.13, we need at most $m + \rho(n) - 1$ additional gates in any of these cases. Hence,

$$\Phi(5,m,n) = \big(\alpha+\phi(5)\big)m + \rho(n) + \gamma \overset{d=5}{=} m + \rho(n) - 1$$

is an upper bound on the number of additional gates, which proves the statement for $d=5$.

Case 2.2.2.2: Assume that $d \geq 6$.

By induction hypothesis, we need at most $\Phi(d-1,k,n)$ additional gates for $\mathrm{C}\big(s,t'\big)$ and at most $\Phi\Big(d-1, m-k, \frac{k-1}{2}\Big)$ additional gates for $\Big(\mathrm{C}\Big(\widehat{t'},t''\Big)\Big)^*$. Note that the concatenation gate is already counted in the case of an alternating split (gate type (ii)). In total, we have at most

$$\begin{aligned}
&\Phi(d-1,k,n) + \Phi\left(d-1, m-k, \frac{k-1}{2}\right) \\
= \quad & \big(\alpha + \phi(d-1)\big)k + \rho(n) + \gamma + \big(\alpha + \phi(d-1)\big)(m-k) + \rho\left(\frac{k-1}{2}\right) + \gamma \\
= \quad & \big(\alpha + \phi(d-1)\big)m + \rho(n) + \rho\left(\frac{k-1}{2}\right) + 2\gamma
\end{aligned}$$

additional gates. We need to show that this is at most $\Phi(d,m,n) = \big(\alpha + \phi(d)\big)m + \rho(n) + \gamma$. Hence, it suffices to show

$$\rho\left(\frac{k-1}{2}\right) + \big(\alpha + \phi(d-1)\big)m + \gamma \leq \big(\alpha + \phi(d)\big)m\,.$$

Since for $d \geq 6$, by Lemma 3.4.17, we have

$$\phi(d) - \phi(d-1) = \psi(d)\,,$$

it remains to show that

$$\rho\left(\frac{k-1}{2}\right) + \gamma \leq \psi(d)m\,. \tag{3.34}$$

Due to assumption (3.33), we have $m > \mu(d-1,0) = \frac{2^{d-1}-2}{d-1} + 2$. As m is integral, this implies

$$m \geq \left\lfloor \frac{2^{d-1}-2}{d-1} \right\rfloor + 3\,.$$

Furthermore, by the choice of k in line 18, k is the maximum odd integer with $k \leq \mu(d-1,n)$, in other words,

$$k = \mathrm{flodd}\big(\mu(d-1,n)\big) = \mathrm{flodd}\left(\frac{2^{d-1}-n-2}{d-1} + 2\right) \leq \mathrm{flodd}\left(\frac{2^{d-1}-2}{d-1}\right) + 2\,.$$

Using these two bounds, the fact that $\psi(d) \geq 0$ for all $d \geq 6$ by its definition in Lemma 3.4.16 and the fact that ρ is increasing by Observation 3.3.16, inequality (*3.34*) is hence implied by

$$\rho\left(\frac{\mathrm{flodd}\left(\frac{2^{d-1}-2}{d-1}\right)+1}{2}\right) + \gamma \leq \psi(d)\left(\left\lfloor \frac{2^{d-1}-2}{d-1}\right\rfloor + 3\right),$$

which holds with equality by definition of ψ for $\gamma = -1$ by Lemma 3.4.16.

Hence, the induction step also holds in case 2.2.2.2 and the lemma is proven. □

Now, we can finally state and prove the main theorem of this section.

Theorem 3.4.19. *Given input variables $s = (s_0, \ldots, s_{n-1})$ and $t = (t_0, \ldots, t_{m-1})$ with $m \geq 2$, Algorithm 3.5 (page 89) computes a circuit $\mathrm{C}(s,t)$ with*

$$\mathrm{size}(\mathrm{C}(s,t)) \leq 4.15m + n + \rho(n) - 4\,,$$

where $\rho(n) = \begin{cases} n & \text{if } n \in \{0,1,2\} \\ \lfloor 2\log_2(n-1) \rfloor & \text{if } n \geq 3 \end{cases}$ *as in Definition 3.3.15.*

Proof. We partition the gates of $\mathrm{C}(s,t)$ as in Definition 3.4.7.

(i) There are at most $m+n-2$ gates contained in the leftist trees S_0 and S_1 by Lemma 3.4.8.

(ii) There are at most $m-1$ alternating-split gates by Corollary 3.4.11.

(iii) - (v) Let $d := d_{\min}(n,m)$. By Lemma 3.4.18, there are at most

$$\begin{aligned}\Phi(d,m,n) &= \big(\alpha+\phi(d)\big)m+\rho(n)+\gamma \\ &\overset{\text{Lem. 3.4.17}}{\leq} \alpha m+\rho(n)+\gamma \\ &\overset{\substack{\alpha=2.15,\\ \gamma=-1}}{=} 2.15m+\rho(n)-1\end{aligned}$$

additional gates.

Adding up all these gates yields the claimed size bound. □

Lemma 3.4.20. *Given symmetric inputs $s=(s_0,\dots,s_{n-1})$ and alternating inputs $t=(t_0,\dots,t_{m-1})$, Algorithm 3.5 (page 89) has running time $\mathcal{O}((m+n)(\log_2 n+\log_2 m))$.*

Proof. Constructing the leftist trees S_0 and S_1 takes time $\mathcal{O}(m+n)$, and computing the data from Lemma 3.3.11 takes time $\mathcal{O}((m+n)\log_2(m+n))$. It remains to bound the running time of Algorithm 3.1 (page 74).

Recall that in each recursive call of Algorithm 3.1, t is a consecutive set of the inputs by Observation 3.4.4 and s is triangular by Lemma 3.4.6, i.e., s consists of two consecutive input sets by Definition 3.3.5. Hence, we can pass on s and t during the algorithm via a constant number of indices.

Note that in each recursive call of Algorithm 3.1, we build at least one gate. Hence, by Theorem 3.4.19 there are at most $\mathcal{O}(m+n)$ recursive calls.

In a single recursive call, the running time is dominated by lines 3 and 18 and the construction of symmetric trees using Theorem 3.3.12. Using binary search, line 18 can be executed in time $\mathcal{O}(\log_2 m)$, and, as by Theorem 3.2.4, we have $d\in\mathcal{O}(\log_2(m+n))$, line 3 can be executed in time $\mathcal{O}(\log_2\log_2(m+n))$. Note that each symmetric tree computed has s as inputs, plus potentially t_0 and t_1. As s and $s \mathbin{+\!\!+} (t_0)$ are both triangular by Lemma 3.4.6, by Theorem 3.3.12, computing a single symmetric tree requires time at most $\mathcal{O}(\log_2 n)$ using the precomputed data from Lemma 3.3.11.

In total, this means that Algorithm 3.1 runs in time $\mathcal{O}((m+n)(\log_2 m+\log_2 n))$. □

For the special case of AND-OR paths, plugging together Corollary 3.2.5, Theorem 3.4.19, and Lemma 3.4.20 yields the following result.

Corollary 3.4.21. *Given input variables $t=(t_0,\dots,t_{m-1})$ with $m\geq 2$, Algorithm 3.5 (page 89) computes a circuit for $g(t)$ with depth at most*

$$\log_2 m+\log_2\log_2 m+1.58$$

and size at most

$$4.15m-4$$

in running time $\mathcal{O}(m\log_2 m)$. □

Chapter 4
Improved Bounds for Delay Optimization

In this chapter, we consider the And-Or Path Circuit Optimization Problem, i.e., we aim at constructing And-Or path circuits with a good delay with respect to prescribed input arrival times. Most theorems and proofs of this chapter have been published previously in more compact form in Brenner and Hermann [BH19].

In this chapter, we will generalize our algorithm from Chapter 3 from depth optimization to delay optimization. We prove that for an And-Or path $g\big((t_0, \dots, t_{m-1})\big)$ with input arrival times $t_0, \dots, t_{m-1} \in \mathbb{N}$, our algorithm computes And-Or path circuits with a delay of at most

$$\log_2 W + \log_2 \log_2 m + \log_2 \log_2 \log_2 m + 4.3\,,$$

where $W := \sum_{i=0}^{m-1} 2^{a(t_i)}$. This is the best known upper bound on the delay of And-Or path circuits known so far. It improves significantly in comparison to the previously best upper bound of

$$\lceil \log_2 W \rceil + 2\sqrt{2 \log_2 m - 1} + 6$$

by Spirkl [Spi14], in particular in comparison to the asymptotic lower bound of $\log_2 m + \log_2 \log_2 m + \text{const}$ for the special case of depth optimization by Commentz-Walter [Com79]. For arbitrary arrival times, $\lceil \log_2 W \rceil$ is the only lower bound known.

Improving the analysis of our algorithm slightly, we decrease the size bound stated in [BH19] from $\mathcal{O}(m \log_2 m \log_2 \log_2 m)$ to $\mathcal{O}(m \log_2 m)$, and the running time bound from $\mathcal{O}(m^2 \log_2 m)$ to $\mathcal{O}(m \log_2^2 m)$.

Recall that in Section 2.6.2, we saw recursive techniques for the construction of And-Or path circuits (see, e.g., Corollaries 2.6.17 and 2.6.18). The most general variants of these recursion strategies use extended And-Or paths (cf. Definition 2.6.14) as intermediate solutions. Thus, just as in [Gri08] and Chapter 3, we shall optimize extended And-Or paths and not only And-Or paths. In these proofs, the depth is estimated by a reverse argument: Given a fixed depth bound d and a fixed number n of symmetric inputs s, one determines how many alternating inputs t an And-Or path may have such that $f(s,t)$ can be realized with depth at most d.

For delay optimization, we shall proceed similarly: Given a fixed delay bound d and symmetric inputs s with a fixed *weight* w, we will determine in Section 4.1 for which alternating inputs t a realization of $f(s,t)$ with delay d can be guaranteed. In

Section 4.2, we will deduce the arising upper bound on the delay of AND-OR path circuits with prescribed arrival times. Finally, in Section 4.3, we will analyze our circuits and our algorithm.

4.1 Bounding the Weight

In this section, we ideally would like to characterize exactly for which alternating inputs t a realization $f(s,t)$ with delay d is possible if d and s are fixed. However, even for small d, the set of these alternating inputs has a very complicated structure Instead, we will thus distinguish different sets of inputs by their weight only, and will bound the maximum weight W such that any alternating inputs t with weight at most W admit realizing $f(s,t)$ with a fixed delay d and symmetric inputs s with a fixed weight w. The goal of this section is proving the following theorem.

Definition 4.1.1. For $\zeta = 1.9$, the function $\nu\colon \mathbb{N}_{\geq 2} \times \mathbb{N} \to \mathbb{R}$ is defined by

$$\nu(d,w) = \zeta \frac{2^{d-1} - w}{d \log_2 d}\,.$$

Theorem 4.1.2. *Let $d, w \in \mathbb{N}$ with $d > 1$ and $0 \leq w < 2^{d-1}$ be given. Consider Boolean input variables s and t with $W(s) = w$ and*

$$W(t) \leq \nu(d,w)\,.$$

Then, we can construct a circuit realizing $f(s,t)$ with delay at most d.

When applied to inputs with all-zero arrival times and hence depth optimization, this theorem says that for $d, n, m \in \mathbb{N}$ with $d \geq 2$, $0 \leq n < 2^{d-1}$ and $m \leq \zeta \frac{2^{d-1}-n}{d \log_2 d}$, we can construct a circuit realizing an AND-OR path with n symmetric inputs and m alternating inputs with depth at most d. Note that this statement is similar to the stronger Proposition 3.1.14: Here, based on [Gri08], we prove that this is even true for $m \leq \frac{2^d - n - 2}{d} + 2$. Up to constants, these bounds thus differ by a factor of $\log_2 d$ which we lose due to the generalization to arbitrary arrival times.

Just as an upper bound on the depth of AND-OR path circuits can be deduced from Proposition 3.1.14, Theorem 4.1.2 will yield an upper bound on the delay of AND-OR path circuits for inputs with prescribed arrival times.

Remark 4.1.3. In Theorem 4.2.4, Theorem 4.1.2 will allow us to deduce the desired upper bound of $\log_2 W + \log_2 \log_2 m + \log_2 \log_2 \log_2 m + 4.3$ on the delay of AND-OR path circuits. In the proof of Theorem 4.2.4, we will see that the choice of the constant ζ influences the additive constant (here 4.3) in the delay bound only. If we chose $\zeta := 1$, we would need to replace the additive constant by 5. An improvement of the additive term to 4.2 would only be possible for $\zeta \geq 1.992$, for which we cannot prove Theorem 4.1.2.

Most parts of the proof of Theorem 4.1.2 will work for any ζ with $1 \leq \zeta < 2$; only in Lemma 4.1.16, we demand $\zeta \leq 1.9$.

We would like to prove Theorem 4.1.2 by induction on d based on the restructuring methods presented in Section 2.6.2, similarly to the proof of Proposition 3.1.14 based on [Gri08]. The main recursion strategy will again be the alternating split with an odd prefix (see Corollary 2.6.17): Given inputs $s = (s_0, \dots, s_{n-1})$ and $t = (t_0, \dots, t_{m-1})$ and an odd integer k with $1 \leq k < m$, we have

$$f(s,t) = f\big(s, (t_0, \dots, t_{k-1})\big) \land f^*\big((t_1, t_3, \dots, t_{k-2}), (t_k, \dots, t_{m-1})\big)\,,$$

or, when recalling the auxiliary notation $t' = (t_0, \dots, t_{k-1})$, $t'' = (t_k, \dots, t_{m-1})$ and $\widehat{t'} = (t_1, t_3, \dots, t_{k-2})$, we have

$$f(s,t) = f\big(s,t'\big) \wedge f^*\Big(\widehat{t'}, t''\Big)\,. \tag{4.1}$$

The following remark indicates why an inductive proof of Theorem 4.1.2 using the alternating split will not work for arbitrary arrival times.

Remark 4.1.4. Suppose that Theorem 4.1.2 holds for some $d \geq 2$ and all $0 \leq w < 2^{d-1}$. In order to prove Theorem 4.1.2 for $d+1$, we need to show that, given input variables s and t with $0 \leq w := W(s) < 2^d$ and $W(t) \leq \nu(d+1, w)$, the extended And-Or path $f(s,t)$ can be realized with delay $d+1$. In the case that $w < 2^{d-1}$, we would like to apply the alternating split (4.1). If we choose the prefix t' of t such that

$$W(t') \leq \nu(d,w) \tag{4.2}$$

(assuming this is possible), the induction hypothesis and the assumption $w < 2^{d-1}$ allow us to construct a circuit for $f(s,t')$ with delay d. Thus, in order to construct a circuit with delay $d+1$ for $f(s,t)$, it remains to prove that $f^*(\widehat{t'}, t'')$ admits a circuit with delay d. Again, by induction hypothesis, we need to show that $W(t'') \leq \nu(d,w)$. But the only fact we know about $W(t'')$ is that

$$W(t'') = W(t) - W(t')\,. \tag{4.3}$$

Even if we choose the prefix t' maximal with (4.2), this will not give us a meaningful upper bound on $W(t'')$ since $W(t')$ might be arbitrarily small in comparison to $W(t)$ due to the presence of arbitrary arrival times.

Note that this is what distinguishes our proof from Grinchuk's [Gri08]: For all-zero arrival times, choosing t' maximal with (4.2) works well, since then, by maximality, we have $W(t') > \nu(d,n) - 2$, hence Equation (4.3) yields

$$W(t'') = W(t) - W(t') < W(t) - \nu(d,n) - 2\,.$$

It turns out that this upper bound on $W(t'')$ suffices to prove that $f^*(\widehat{t'}, t'')$ can be realized with delay d.

When arbitrary arrival times are present, a different proof idea is needed.

As a consequence, instead of directly proving Theorem 4.1.2 by induction, we strengthen the induction hypothesis and prove the stronger Theorem 4.1.6.

Definition 4.1.5. Let $m \in \mathbb{N}$. For inputs $t = (t_0, \dots, t_{m-1})$ with arrival times $a(t_0), \dots, a(t_{m-1}) \in \mathbb{N}$, we denote by Λ_t the weight of the last two (or fewer) entries of t, i.e.,

$$\Lambda_t := \begin{cases} 0, & m = 0\,, \\ W(t_0), & m = 1\,, \\ W(t_{m-2}) + W(t_{m-1}), & m > 1\,. \end{cases}$$

Theorem 4.1.6. *Let $d, w \in \mathbb{N}$ with $d \geq 2$ and $0 \leq w < 2^{d-1}$ be given. Consider Boolean input variables s and t with $W(s) = w$ and*

$$W(t) \leq \nu(d,w) + \frac{d-1}{d}\Lambda_t\,. \tag{4.4}$$

Then, we can construct a circuit realizing $f(s,t)$ with delay at most d.

Note that since $\Lambda_t \geq 0$, Theorem 4.1.6 implies Theorem 4.1.2. The proof of Theorem 4.1.6 is the most important step to prove the new delay guarantee and covers the rest of this section.

First, we note how requirement (4.4) can be rewritten.

Observation 4.1.7. Consider $d, w \in \mathbb{N}$ with $d \geq 2$ and $0 \leq w < 2^{d-1}$ and Boolean input variables s and t with $W(s) = w$. The following three statements are equivalent:

$$W(t) \leq \nu(d,w) + \frac{d-1}{d}\Lambda_t$$

$$\sum_{i=0}^{m-3} W(t_i) + \frac{\Lambda_t}{d} \leq \nu(d,w) \tag{4.5}$$

$$d\sum_{i=0}^{m-3} W(t_i) + \Lambda_t \leq d\nu(d,w) \tag{4.6}$$

Note that the sums in Observation 4.1.7 are empty in case $m \leq 2$. Nevertheless, the three statements are equivalent.

Next, we give an upper bound on $W(t) + w$ in the setting of Theorem 4.1.6.

Lemma 4.1.8. *Assuming the conditions of Theorem 4.1.6, we have*

$$W(t) + w \leq \begin{cases} 2^{d-1} & \text{if } d \geq 2^{\zeta}\,, \\ \frac{2^d}{\log_2 d} & \text{otherwise}\,. \end{cases} \tag{4.7}$$

Proof. Using inequality (4.6), we obtain

$$W(t)+w \overset{(4.6)}{\leq} d\nu(d,w)+w \overset{\text{Def. 4.1.1}}{=} d\zeta\frac{2^{d-1}-w}{d\log_2 d}+w = \frac{\zeta 2^{d-1} + (\log_2 d - \zeta)w}{\log_2 d}\,. \tag{4.8}$$

If $d \geq 2^{\zeta}$, along with the condition $w < 2^{d-1}$, this implies

$$W(t) + w \overset{(4.8)}{\leq} \frac{\zeta 2^{d-1} + (\log_2 d - \zeta)w}{\log_2 d} \leq \frac{2^{d-1}(\zeta + \log_2 d - \zeta)}{\log_2 d} = 2^{d-1}\,.$$

Otherwise, if $d < 2^{\zeta}$, the condition $w \geq 0$ yields

$$W(t) + w \overset{(4.8)}{\leq} \frac{\zeta 2^{d-1} + (\log_2 d - \zeta)w}{\log_2 d} \leq \frac{\zeta 2^{d-1}}{\log_2 d} \overset{\zeta<2}{<} \frac{2^d}{\log_2 d}\,. \qquad \square$$

The equivalent requirements (4.4), (4.5) and (4.6) as well as Lemma 4.1.8 will be used extensively when proving Theorem 4.1.6. In this proof, we proceed by induction on d. In Lemmas 4.1.9 and 4.1.10, we will show as a base case that Theorem 4.1.6 holds for $d \leq 3$ and $m \leq 2$. Then, in Theorem 4.1.11, we will prove the inductive step: Assuming that Theorem 4.1.6 holds for some $d \geq 3$ and all $0 \leq w < 2^{d-1}$, we will prove the statement for $d+1$ and all $0 \leq w < 2^d$ and $m \geq 3$.

Lemma 4.1.9. *Assuming the conditions of Theorem 4.1.6 for $m \leq 2$, we can construct a circuit for $f(s,t)$ with delay d.*

Proof. Recall from Observation 2.6.21 that for $m \leq 2$, the function $f(s,t)$ is a symmetric binary tree which can be realized with delay d by Huffman coding if and only if $W(t) + w \leq 2^d$. But this holds by inequality (4.7). $\square$

Lemma 4.1.10. *Assuming the conditions of Theorem 4.1.6 for $d = 2, 3$, we can construct a circuit for $f(s,t)$ with delay d.*

Proof. The case $m \le 2$ is already covered by Lemma 4.1.9.

Now let $m \ge 3$. Requirement (4.5) yields

$$\sum_{i=0}^{m-3} W(t_i) + \frac{\Lambda_t}{d} \overset{(4.5)}{\le} \nu(d,w) = \zeta \frac{2^{d-1} - w}{d \log_2 d} \overset{\zeta<2}{<} \frac{2^d - 2w}{d \log_2 d}\,.$$

Note that $W(t_i) \ge 1$ for all $i \in \{0, \dots, m-1\}$. For $d = 2$, these two observations lead to the contradiction

$$2 \overset{m\ge3}{\le} m - 2 + \frac{2}{2} \overset{\substack{m\ge3,\\ W(t_i)\ge1}}{\le} \sum_{i=0}^{m-3} W(t_i) + \frac{\Lambda_t}{2} \overset{4.1.10}{<} \frac{4-2w}{2 \cdot 1} \overset{w\ge0}{\le} 2\,,$$

i.e., for $d = 2$, we always have $m \le 2$ and have already proven the required statement.

Similarly, if $d = 3$, we obtain

$$1 + \frac{2}{3} \overset{m\ge3}{\le} m - 2 + \frac{2}{3} \overset{\substack{m\ge3,\\ W(t_i)\ge1}}{\le} \sum_{i=0}^{m-3} W(t_i) + \frac{\Lambda_t}{3} \overset{4.1.10}{<} \frac{8-2w}{3 \cdot \log_2(3)} < \begin{cases} 2 & w = 0 \\ \frac{4}{3} & w \ge 1 \end{cases}.$$

If $w \ge 1$, this is a contradiction. For $w = 0$, the only case where this is no contradiction is $m = 3$ with arrival times $a(t_0) = a(t_1) = a(t_2) = 0$. With these arrival times, $f(s,t) = t_0 \wedge (t_1 \vee t_2)$ yields a circuit with delay $2 < 3$. □

Theorem 4.1.11. *Assume inductively that for some $d \ge 3$ and all $0 \le w < 2^{d-1}$, Theorem 4.1.6 holds. Then, given inputs s and t with $w := W(s)$ such that $0 \le w < 2^d$, $m \ge 3$ and*

$$W(t) \le \nu(d+1, w) + \frac{d}{d+1}\Lambda_t\,, \tag{4.9}$$

we can construct a circuit for $f(s,t)$ with delay at most $(d+1)$.

First, we prove the following auxiliary lemma.

Lemma 4.1.12. *Assuming the conditions of Theorem 4.1.11, we have*

$$\nu(d,0) + \frac{d-1}{d}\Lambda_t - \nu(d+1,w) - \frac{d}{d+1}\Lambda_t \ge \zeta \frac{2^{d-1}\log_2(d+1) - (2^d - w)\log_2 d}{d \log_2 d \log_2(d+1)}\,.$$

Proof. Using the bound on Λ_t implied by inequality (4.6), using $d+1$ instead of d, we calculate

$$\begin{aligned}
&\nu(d,0) + \frac{d-1}{d}\Lambda_t - \nu(d+1,w) - \frac{d}{d+1}\Lambda_t \\
\overset{\text{Def. 4.1.1}}{=}\ & \zeta\frac{2^{d-1}}{d\log_2 d} - \zeta\frac{2^d - w}{(d+1)\log_2(d+1)} + \frac{d^2-1-d^2}{d(d+1)}\Lambda_t \\
=\ & \zeta\frac{2^{d-1}(d+1)\log_2(d+1) - (2^d-w)d\log_2 d}{d(d+1)\log_2 d\log_2(d+1)} - \frac{1}{d(d+1)}\Lambda_t \\
\overset{(4.6)}{\geq}\ & \zeta\frac{2^{d-1}(d+1)\log_2(d+1) - (2^d-w)d\log_2 d}{d(d+1)\log_2 d\log_2(d+1)} - \frac{1}{d(d+1)}(d+1)\nu(d+1,w) \\
\overset{\text{Def. 4.1.1}}{=}\ & \zeta\frac{2^{d-1}(d+1)\log_2(d+1) - (2^d-w)d\log_2 d}{d(d+1)\log_2 d\log_2(d+1)} - \frac{1}{d}\zeta\frac{2^d-w}{(d+1)\log_2(d+1)} \\
=\ & \zeta\frac{2^{d-1}(d+1)\log_2(d+1) - (2^d-w)d\log_2 d - (2^d-w)\log_2 d}{d(d+1)\log_2 d\log_2(d+1)} \\
=\ & \zeta\frac{2^{d-1}\log_2(d+1) - (2^d-w)\log_2 d}{d\log_2 d\log_2(d+1)}\,. \qquad \square
\end{aligned}$$

This is the only ingredient needed to prove Theorem 4.1.11 for the case that $2^{d-1} \leq w < 2^d$. In this case, the weight w of the symmetric inputs s is so large that we already need delay d to construct a symmetric tree on s. Hence, intuitively, we do not waste a lot if we use a symmetric split that constructs the circuits for s and t independently as in Equation (4.10). The following lemma proves that this indeed yields a realization with delay $d+1$.

Lemma 4.1.13. *Theorem 4.1.11 holds for all w satisfying $2^{d-1} \leq w < 2^d$.*

Proof. The symmetric split (2.33) yields the realization

$$f(s,t) = \operatorname{sym}(s) \wedge g(t)\,. \tag{4.10}$$

Since $w < 2^d$, Theorem 2.3.21 allows the construction of a symmetric tree on inputs s with delay d. Thus, it remains to show that $f((),t) = g(t)$ can be realized by a circuit with delay d. By induction hypothesis, this is true if $W(t) \leq \nu(d,0) + \frac{d-1}{d}\Lambda_t$. In order to show this, as $W(t) \leq \nu(d+1,w) + \frac{d}{d+1}\Lambda_t$ by (4.9), it suffices to show

$$\nu(d+1,w) + \frac{d}{d+1}\Lambda_t \leq \nu(d,0) + \frac{d-1}{d}\Lambda_t\,.$$

Subtracting the left-hand side from the right-hand side, we prove this via

$$\begin{aligned}
&\nu(d,0) + \frac{d-1}{d}\Lambda_t - \nu(d+1,w) - \frac{d}{d+1}\Lambda_t \\
\overset{\text{Lem. 4.1.12}}{\geq}\ & \zeta\frac{2^{d-1}\log_2(d+1) - (2^d-w)\log_2 d}{d\log_2 d\log_2(d+1)} \\
\overset{w\geq 2^{d-1}}{\geq}\ & \zeta\frac{2^{d-1}(\log_2(d+1) - \log_2 d)}{d\log_2 d\log_2(d+1)} \\
\geq\ & 0\,.
\end{aligned}$$

Hence, the symmetric split (4.10) yields a realization for $f(s,t)$ with delay $d+1$. $\square$

In the case $0 \leq w < 2^{d-1}$, we need a bound on the difference of the logarithms of two consecutive integers.

Observation 4.1.14. For $d \geq 3$, we have $d \geq \ln(2)(d+1)$ and thus

$$\log_2(d+1) - \log_2 d = \frac{\ln(d+1) - \ln d}{\ln(2)} = \int_d^{d+1} \frac{1}{\ln(2)x} dx \geq \frac{1}{\ln(2)(d+1)} \geq \frac{1}{d}.$$

Furthermore, for simplicity, we note two direct consequences of Equation (4.7) and Equation (4.5).

Lemma 4.1.15. *Assuming the conditions of Theorem 4.1.11, we have*

$$W(t) + w \leq 2^d \tag{4.11}$$

and

$$\sum_{i=0}^{m-3} W(t_i) < 2^{d-1}. \tag{4.12}$$

Proof. Inequality (4.11) is directly implied by Equation (4.7) since $d+1 \geq 4 > 2^\zeta$. For proving inequality (4.12), we compute

$$\sum_{i=0}^{m-3} W(t_i) \overset{(4.5)}{\leq} \zeta \frac{2^d - w}{(d+1)\log_2(d+1)} \overset{\substack{w\geq 0\\ \zeta<2}}{<} \frac{2^{d+1}}{(d+1)\log_2(d+1)} \overset{d\geq 2}{\leq} 2^{d-1}. \qquad \square$$

Moreover, we will need the following technical lemma.

Lemma 4.1.16. *For $d \in \mathbb{N}$, $d \geq 3$, we have*

$$2^d \log_2 d - \left(\log_2 d + \frac{1}{d}\right)\zeta 2^{d-1} + \left(2 + \frac{1}{\zeta} d \log_2 d\right) \log_2 d \log_2(d+1) \geq 0.$$

Proof. Note that for $d \geq 7$, the statement is already implied by

$$2^d \log_2 d - \left(\log_2 d + \frac{1}{d}\right)\zeta 2^{d-1} \overset{\zeta=1.9}{=} 2^{d-1}\left(0.1 \log_2 d - \frac{1.9}{d}\right) \overset{d\geq 7}{\geq} 0.$$

For $3 \leq d \leq 6$, we have

$$\begin{aligned}
& 2^d \log_2 d - \left(\log_2 d + \frac{1}{d}\right)\zeta 2^{d-1} + \left(2 + \frac{1}{\zeta} d \log_2 d\right) \log_2 d \log_2(d+1) \\
\overset{\zeta=1.9}{=} \; & \log_2 d \left(0.1 \cdot 2^{d-1} + \left(2 + \frac{1}{1.9} d \log_2 d\right) \log_2(d+1)\right) - \frac{1.9}{d} 2^{d-1} \\
\overset{d\geq 3}{\geq} \; & \log_2(3) \left(0.1 \cdot 2^2 + \left(2 + \frac{1}{1.9} 3 \log_2(3)\right) 2\right) - \frac{1.9}{d} 2^{d-1} \\
> \; & \log_2(3)(0.4 + 4.5 \cdot 2) - \frac{1.9}{d} 2^{d-1} \\
> \; & 14 - \frac{1.9}{d} 2^{d-1}.
\end{aligned}$$

As the function $x \mapsto \frac{2^{x-1}}{x}$ is monotonely increasing for $x \geq 2$ by Lemma 3.2.2, the value $14 - \frac{1.9}{6} \cdot 2^{6-1}$ is a lower bound on the value of the function $d \mapsto 14 - \frac{1.9}{d} 2^{d-1}$ for $3 \leq d \leq 6$. As $14 - \frac{1.9}{6} \cdot 2^{6-1} > 3 > 0$, this proves the lemma. $\square$

Now, we will prove Theorem 4.1.11 for the case that $0 \leq w < 2^{d-1}$. Here, we will use that w is small enough such that the induction hypothesis allows to realize $f(s,t')$ with delay d for some prefix t' of t. Based on this, we will show that the alternating split (2.28) will yield a realization for $f(s,t)$.

Lemma 4.1.17. *Theorem 4.1.11 holds for each w satisfying $0 \leq w < 2^{d-1}$.*

Proof. We prove this lemma via a case distinction. By assumption, we have $m \geq 3$.

In case 2, we will consider a prefix t' of the inputs t with weight at most $\nu(d,w)$ in order to proceed similarly as indicated in Remark 4.1.4 and before this lemma. If the weight of t_0 is already larger than this, such a prefix does not exist. We deal with this situation in case 1.

Case 1: Assume that

$$W(t_0) > \nu(d,w)\,. \tag{4.13}$$

The alternating split (2.28) applied with prefix-length $k = 1$ yields

$$f(s,t) = f\big(s,t'\big) \wedge f^*\Big(\widehat{t'},t''\Big) = f\big(s,(t_0)\big) \wedge f^*\big((),(t_1,\dots,t_{m-1})\big)\,. \tag{4.14}$$

By inequality (4.11), we have $W(t_0) + w \leq W(t) + w \leq 2^d$. Hence, by Theorem 2.3.21, the symmetric tree $f\big(s,(t_0)\big)$ can be realized with delay d. Thus, it remains to check inductively that $f^*\big((),(t_1,\dots,t_{m-1})\big)$ can be realized by a circuit with delay d. For this, note that requirement (4.9) and condition (4.13) imply

$$W((t_1,t_2,\dots,t_{m-1})) = W(t) - W(t_0) < \nu(d+1,w) + \frac{d}{d+1}\Lambda_t - \nu(d,w)\,,$$

which we claim to be at most $\nu(d,0) + \frac{d-1}{d}\Lambda_t$. This can be shown by

$$\begin{aligned}
&\nu(d,0) + \frac{d-1}{d}\Lambda_t - \nu(d+1,w) - \frac{d}{d+1}\Lambda_t + \nu(d,w)\\
\overset{\text{Lem. 4.1.12}}{\geq}\;& \zeta\frac{2^{d-1}\log_2(d+1) - (2^d-w)\log_2 d}{d\log_2 d\log_2(d+1)} + \zeta\frac{2^{d-1}-w}{d\log_2 d}\\
=\;& \zeta\frac{2^{d-1}\log_2(d+1) - (2^d-w)\log_2 d + (2^{d-1}-w)\log_2(d+1)}{d\log_2 d\log_2(d+1)}\\
=\;& \zeta\frac{(2^d-w)(\log_2(d+1)-\log_2 d)}{d\log_2 d\log_2(d+1)}\\
\overset{w<2^d}{\geq}\;& 0\,.
\end{aligned}$$

Thus, realization (4.14) yields a delay of $d+1$ for $f(s,t)$, which proves the lemma for the case that $W(t_0) > \nu(d,w)$.

Case 2: Assume that $W(t_0) \leq \nu(d,w)$.

Therefore, we may consider a maximum odd-length prefix $t' = (t_0,t_1,\dots,t_{k-1})$ of t with $0 \leq k \leq m$ odd such that

$$W\big(t'\big) \leq \nu(d,w)\,. \tag{4.15}$$

We define $t'' := (t_k,\dots,t_{m-1})$.

If t'' is empty, there is nothing to show since, by induction hypothesis, we can construct a circuit for $f(s,t) = f(s,t')$ with a delay of $d < d+1$ due to $w < 2^{d-1}$.

Otherwise, we will realize $f(s,t)$ by a circuit with delay $d+1$ using the alternating split (2.28) for some prefix t^* of t to be determined, i.e.,

$$f(s,t) = f\big(s,t^*\big) \wedge f^*\Big(\widehat{t^*}, t^{**}\Big), \tag{4.16}$$

where $t^* = (t_0, t_1, \dots, t_{l-1})$ for some $0 \leq l < m-1$, $t^{**} := (t_l, \dots, t_{m-1})$ and $\widehat{t^*} = \Big(t_1^*, t_3^*, \dots, t_{l-2}^*\Big) = (t_1, t_3, \dots, t_{l-2})$. The rough idea is to choose $t^* = t'$, and to add the first two inputs of t'' to t^* if their weight is small. Our main argument, which is presented in case 2.2.2, requires that $\{t_k, t_{k+1}\} \cap \{t_{m-2}, t_{m-1}\} = \emptyset$, i.e., that t'' has at least 4 elements. Thus, in case 2.1, we handle the t'' with at most 2 elements, and in case 2.2.1 those with exactly 3 elements.

Case 2.1: Assume that $|t''| \leq 2$.

We set $t^* := t'$, thus $t^{**} = t''$. By induction hypothesis and due to $w < 2^{d-1}$, inequality (4.15) allows us to realize $f(s,t^*)$ by a circuit with delay d. Since t^* has at most 2 elements, by Observation 2.6.21, we can realize $f^*\Big(\widehat{t^*}, t^{**}\Big)$ as a binary tree with delay d since $W\Big(\widehat{t^*}\Big) + W(t^{**}) \leq W(t) \overset{(4.11)}{\leq} 2^d$.

Case 2.2: Assume that $|t''| \geq 3$.

Let $\tilde{t} := (t_0, \dots, t_{k+1})$. We need to find an appropriate prefix t^* of t for realization (4.16) such that both $f(s,t^*)$ and $f^*(\widehat{t^*}, t^{**})$ can be realized by a circuit with delay d by induction hypothesis. We choose t^* depending on the weight of $\tilde{t}$:

(a) If $W\Big(\tilde{t}\Big) \leq \nu(d,w) + \frac{d-1}{d}\Lambda_{\tilde{t}}$, we set $t^* := \tilde{t}$.

(b) Otherwise, we set $t^* := t'$. Note that in this case, we particularly have

$$W(t^*) = W(t') = W\Big(\tilde{t}\Big) - \Lambda_{\tilde{t}} > \nu(d,w) + \frac{d-1}{d}\Lambda_{\tilde{t}} - \Lambda_{\tilde{t}} = \nu(d,w) - \frac{1}{d}\Lambda_{\tilde{t}}\,.$$

Figure 4.1 visualizes the case distinction. In either case, the weight of t^* will be of the form

$$W(t^*) = \nu(d,w) + \delta \text{ with } -\frac{1}{d}\Lambda_{\tilde{t}} \leq \delta \leq \frac{d-1}{d}\Lambda_{\tilde{t}}\,. \tag{4.17}$$

The function $f(s,t^*)$ can be realized by a circuit with delay d by induction hypothesis due to $w < 2^{d-1}$ and, in case (a), the upper bound on δ, and in case (b), the choice of t^* such that (4.15) is fulfilled. Consequently, in either case, it remains to show that $f^*\Big(\widehat{t^*}, t^{**}\Big)$ can be realized by a circuit with delay d.

The case that $|t''| = 3$ still needs to be treated separately.

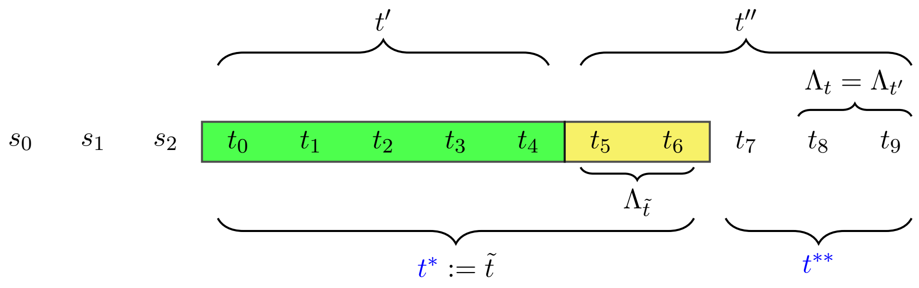
Case 2.2.1: Assume that $|t''| = 3$.

Here, case (a) is easy since we have $t^{**} = (t_{m-1})$, hence $f^*\Big(\widehat{t^*}, t^{**}\Big)$ is a binary tree which can be realized with delay d by Huffman coding since $W(t) \leq 2^d$ due to inequality (4.11).

In case (b), the following claim constructs a realization for $f^*\Big(\widehat{t^*}, t^{**}\Big)$, and Figure 4.2 illustrates the current setting.

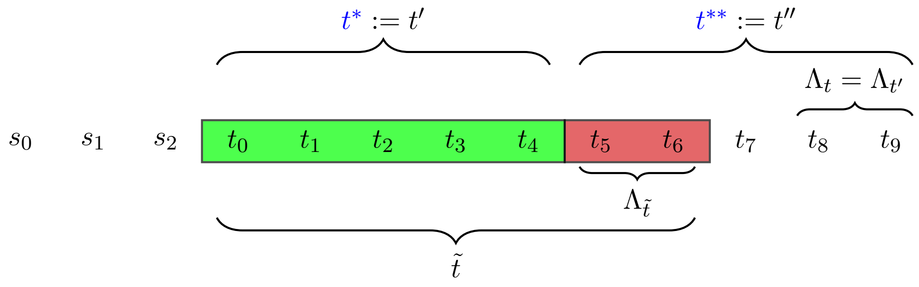
Claim 1. Given that $|t''| = 3$ and

$$W\Big(\tilde{t}\Big) > \nu(d,w) + \frac{d-1}{d}\Lambda_{\tilde{t}}\,, \tag{4.18}$$



t'
t''
$\Lambda_t = \Lambda_{t'}$
s_0 s_1 s_2 t_0 t_1 t_2 t_3 t_4 t_5 t_6 t_7 t_8 t_9
$\Lambda_{\tilde{t}}$
$t^* := \tilde{t}$
t^{**}

(a) In the case that $W\left(\tilde{t}\right) \leq \nu(d,w) + \frac{d-1}{d}\Lambda_{\tilde{t}}$, we set $t^* := \tilde{t}$.



$t^* := t'$
$t^{**} := t''$
$\Lambda_t = \Lambda_{t'}$
s_0 s_1 s_2 t_0 t_1 t_2 t_3 t_4 t_5 t_6 t_7 t_8 t_9
$\Lambda_{\tilde{t}}$
$\tilde{t}$

(b) In the case that $W\left(\tilde{t}\right) > \nu(d,w) + \frac{d-1}{d}\Lambda_{\tilde{t}}$, we set $t^* := t'$.

Figure 4.1: Illustration of the choice of t^*.

the realization

$$f^*\left(\widehat{t'}, t''\right) = \left(\widehat{t'} \vee t_k\right) \vee \left(t_{k+1} \wedge t_{k+2}\right)$$

yields delay d.

Proof of claim: We show that the two sub-formulas $\widehat{t'} \vee t_k$ and $t_{k+1} \wedge t_{k+2}$ both can be realized by a circuit with delay $d-1$. A binary tree $\widehat{t'} \vee t_k$ with delay $d-1$ can be found using Theorem 2.3.21 since

$$W\left(\widehat{t'}\right) + W(t_k) \overset{|t''| \geq 3}{\leq} \sum_{i=0}^{m-3} W(t_i) \overset{(4.12)}{<} 2^{d-1}.$$

As $\Lambda_t = W(t_{k+1}) + W(t_{k+2})$, it remains to show $\Lambda_t \leq 2^{d-1}$, so assume on the contrary that

$$\Lambda_t > 2^{d-1}. \tag{4.19}$$

Due to $d+1 \geq 4$, we may apply (4.7) and have $2^{d-1} \overset{(4.19)}{<} \Lambda_t \overset{(4.7)}{<} 2^d$. First assume that

$$W(t_{k+2}) \geq W(t_{k+1}), \tag{4.20}$$

which implies

$$W(t_{k+2}) = 2^{d-1} \quad \text{and} \quad W(t_{k+1}) \leq 2^{d-2}. \tag{4.21}$$

Note that $\Lambda_{\tilde{t}} = W(t_k) + W(t_{k+1})$.

By combining

$$W(t) - \Lambda_t \overset{(4.9)}{\leq} \nu(d+1,w) - \frac{\Lambda_t}{d+1} = \nu(d+1,w) - \frac{W(t_{k+1}) + W(t_{k+2})}{d+1}$$

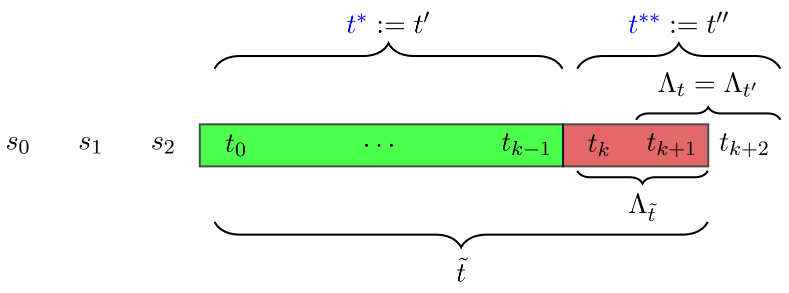


Figure 4.2: Illustration of the setting of Claim 1, where we have $|t''| = 3$ and $W\left(\tilde{t}\right) > \nu(d,w) + \frac{d-1}{d}\Lambda_{\tilde{t}}$.

and

$$\begin{aligned} W(t) - \Lambda_t &= W\left(\tilde{t}\right) - W(t_{k+1}) \\ &\overset{(4.18)}{>} \nu(d,w) + \frac{d-1}{d}\Lambda_{\tilde{t}} - W(t_{k+1}) \\ &= \nu(d,w) + \frac{d-1}{d}\big(W(t_k) + W(t_{k+1})\big) - W(t_{k+1}) \\ &\overset{W(t_k)\geq 1}{\geq} \nu(d,w) + \frac{d-1}{d} - \frac{W(t_{k+1})}{d}, \end{aligned}$$

we obtain

$$\begin{aligned} 0 &< \nu(d+1,w) - \nu(d,w) - \frac{W(t_{k+1}) + W(t_{k+2})}{d+1} - \frac{d-1}{d} + \frac{W(t_{k+1})}{d} \\ &= \zeta\frac{2^d - w}{(d+1)\log_2(d+1)} - \zeta\frac{2^{d-1} - w}{d\log_2 d} + \frac{W(t_{k+1}) - dW(t_{k+2}) - d^2 + 1}{d(d+1)} \\ &= \zeta\frac{(2^d - w)d\log_2 d - (2^{d-1} - w)(d+1)\log_2(d+1)}{d(d+1)\log_2 d\log_2(d+1)} \\ &\quad + \frac{W(t_{k+1}) - dW(t_{k+2}) - d^2 + 1}{d(d+1)} \\ &\overset{w<2^{d-1}}{<} \zeta\frac{2^{d-1}}{(d+1)\log_2(d+1)} + \frac{W(t_{k+1}) - dW(t_{k+2}) - d^2 + 1}{d(d+1)} \\ &\overset{(4.21)}{\leq} \zeta\frac{2^{d-1}}{(d+1)\log_2(d+1)} + \frac{2^{d-2} - d2^{d-1} - d^2 + 1}{d(d+1)} \\ &\overset{\zeta<2}{<} \frac{d2^d + \log_2(d+1)\left(2^{d-2} - d2^{d-1} - d^2 + 1\right)}{d(d+1)\log_2(d+1)} \\ &< 0, \end{aligned}$$

where the last step can be verified explicitly for $d = 3$, and for $d \geq 4$ is implied by

$$\begin{aligned}
d2^d + \log_2(d+1)\Big(2^{d-2} - d2^{d-1}\Big) &= 2^{d-2}\big(4d + \log_2(d+1)(1-2d)\big) \\
&\overset{d\geq 4}{\leq} 2^{d-2}\big(4d + \log_2(5)(1-2d)\big) \\
&= 2^{d-2}\Big(d\big(4 - 2\log_2(5)\big) + \log_2(5)\Big) \\
&< 2^{d-2}\big(-0.64d + \log_2(5)\big) \\
&\overset{d\geq 4}{\leq} 2^{d-2}\big(-2.56 + \log_2(5)\big) \\
&< 0\,.
\end{aligned}$$

This is a contradiction, which proves the claim. □

If assumption (4.20) is not fulfilled, we exchange the inputs t_{k+1} and t_{k+2}, which leaves the function $f(s,t)$ and Λ_t unchanged. Now, we apply the lemma to the modified instance. Note that we are again in case 2 and have $|t''| \leq 3$. As $\Lambda_t > 2^{d-1}$ still holds and now, assumption (4.20) is fulfilled, case 2.2.1 (b) is excluded. Hence, we must be in case 2.1 or 2.2.1 (a) and a realization for $f(s,t)$ with delay at most $d+1$ is provided by the proof in the respective case. See Equation (4.27) for the arising realization of $f(s,t)$.

Now, we may assume that t'' contains at least 4 elements.

Case 2.2.2: Assume that $|t''| \geq 4$.

In particular, we have $\{t_k, t_{k+1}\} \cap \{t_{m-2}, t_{m-1}\} = \emptyset$. Note that since t^* does not contain any of the last two elements of t, we have $W(\widehat{t^*}) \overset{(4.12)}{<} 2^{d-1}$ and thus, by induction hypothesis, it suffices to prove that

$$W(t^{**}) \leq \nu(d, W(\widehat{t^*})) + \frac{d-1}{d}\Lambda_{t^{**}}\,. \tag{4.22}$$

We have

$$W(t^{**}) = W(t) - W(t^*) \overset{(4.9)}{\leq} \nu(d+1, w) + \frac{d}{d+1}\Lambda_t - W(t^*)\,.$$

Since $W(\widehat{t^*}) \leq W(t^*)$ and $\nu(\_,\_)$ is monotonely decreasing in the second parameter by Definition 4.1.1, we have $\nu(d, W(t^*)) \leq \nu(d, W(\widehat{t^*}))$. Furthermore, we have $\Lambda_{t^{**}} = \Lambda_t$. Inequality (*4.22*) is thus implied if we prove the following claim.

Claim 2. We have $\nu(d, W(t^*)) + \frac{d-1}{d}\Lambda_t - \nu(d+1, w) - \frac{d}{d+1}\Lambda_t + W(t^*) \geq 0\,.$

Proof of claim: Here, we use that, as we are in case 2.2.2, we have $\{t_k, t_{k+1}\} \cap \{t_{m-2}, t_{m-1}\} = \emptyset$, and that these four inputs are not contained in t'. Hence, we have

$$W(t') + \Lambda_{\tilde{t}} + \frac{\Lambda_t}{d+1} \overset{\text{case 2.2.2}}{\leq} W(t) \overset{(4.5)}{\leq} \nu(d+1, w)\,. \tag{4.23}$$

We first only bound the summands in the claim that depend on $W(t^*)$ or Λ_t:

$$\begin{aligned}
& -\zeta\frac{W(t^*)}{d\log_2 d} + \frac{d-1}{d}\Lambda_t - \frac{d}{d+1}\Lambda_t + W(t^*) \\
= \quad & W(t^*)\left(1 - \frac{\zeta}{d\log_2 d}\right) - \frac{1}{d(d+1)}\Lambda_t \\
\overset{\substack{d\log_2 d>\zeta,\\(4.17)}}{\geq} \quad & \left(\nu(d,w) - \frac{1}{d}\Lambda_{\tilde{t}}\right)\frac{d\log_2 d - \zeta}{d\log_2 d} - \frac{1}{d(d+1)}\Lambda_t \\
\overset{\text{Def. 4.1.1}}{=} \quad & \zeta\frac{(2^{d-1}-w)(d\log_2 d-\zeta)}{d^2\log_2^2 d} - \frac{d\log_2 d-\zeta}{d^2\log_2 d}\Lambda_{\tilde{t}} - \frac{1}{d(d+1)}\Lambda_t \\
= \quad & \zeta\frac{(2^{d-1}-w)(d\log_2 d-\zeta)}{d^2\log_2^2 d} - \frac{1}{d}\left(\Lambda_{\tilde{t}} + \frac{\Lambda_t}{d+1}\right) + \frac{\zeta\Lambda_{\tilde{t}}}{d^2\log_2 d} \\
\overset{(4.23)}{\geq} \quad & \zeta\frac{(2^{d-1}-w)(d\log_2 d-\zeta)}{d^2\log_2^2 d} - \frac{1}{d}\big(\nu(d+1,w) - W(t')\big) + \frac{\zeta\Lambda_{\tilde{t}}}{d^2\log_2 d} \\
\overset{\substack{\text{Def. 4.1.1,}\\W(t')\geq 1}}{\geq} \quad & \zeta\frac{(2^{d-1}-w)(d\log_2 d-\zeta)}{d^2\log_2^2 d} - \zeta\frac{2^d-w}{d(d+1)\log_2(d+1)} + \frac{1}{d} + \frac{2\zeta}{d^2\log_2 d}
\end{aligned} \tag{4.24}$$

Based on inequality (4.24), the left-hand side of the inequality in the claim can be bounded from below by

$$\begin{aligned}
& \nu(d,W(t^*)) + \frac{d-1}{d}\Lambda_t - \nu(d+1,w) - \frac{d}{d+1}\Lambda_t + W(t^*) \\
\overset{\text{Def. 4.1.1}}{=} \quad & \zeta\frac{2^{d-1}-W(t^*)}{d\log_2 d} + \frac{d-1}{d}\Lambda_t - \zeta\frac{2^d-w}{(d+1)\log_2(d+1)} - \frac{d}{d+1}\Lambda_t + W(t^*) \\
\overset{(4.24)}{\geq} \quad & \zeta\Bigg(\frac{(2^{d-1}-w)(d\log_2 d-\zeta)}{d^2\log_2^2 d} - \frac{2^d-w}{d(d+1)\log_2(d+1)} + \frac{1}{\zeta d} + \frac{2}{d^2\log_2 d} \\
& \qquad + \frac{2^{d-1}}{d\log_2 d} - \frac{2^d-w}{(d+1)\log_2(d+1)}\Bigg) \\
= \quad & \zeta\left(\frac{2^d-w}{d\log_2 d} - \zeta\frac{2^{d-1}-w}{d^2\log_2^2 d} - \frac{(2^d-w)(d+1)}{d(d+1)\log_2(d+1)} + \frac{1}{\zeta d} + \frac{2}{d^2\log_2 d}\right) \\
= \quad & \frac{\zeta}{d^2\log_2^2 d\log_2(d+1)}\Bigg(\log_2(d+1)\Big((2^d-w)d\log_2 d - \zeta(2^{d-1}-w)\Big) \\
& \qquad - (2^d-w)d\log_2^2 d + \left(2 + \frac{1}{\zeta}d\log_2 d\right)\log_2 d\log_2(d+1)\Bigg),
\end{aligned} \tag{4.25}$$

which is required to be non-negative.

Using the bound $\log_2(d+1) \geq \log_2 d + \frac{1}{d}$ stated in Observation 4.1.14, we obtain

$$\begin{aligned}
&\log_2(d+1)\Big((2^d - w)d\log_2 d - \zeta(2^{d-1} - w)\Big) - (2^d - w)d\log_2^2 d \\
\overset{\substack{\zeta < d\log_2 d, \\ \text{Obs. 4.1.14}}}{\geq} &\left(\log_2 d + \frac{1}{d}\right)\Big((2^d - w)d\log_2 d - \zeta(2^{d-1} - w)\Big) - (2^d - w)d\log_2^2 d \\
= \quad &(2^d - w)\log_2 d - \left(\log_2 d + \frac{1}{d}\right)\zeta(2^{d-1} - w) \\
\overset{\zeta \geq 1, w \geq 0}{\geq} &2^d \log_2 d - \left(\log_2 d + \frac{1}{d}\right)\zeta 2^{d-1}\,. \qquad (4.26)
\end{aligned}$$

With this inequality, we have

$$\begin{aligned}
&\log_2(d+1)\Big((2^d - w)d\log_2 d - \zeta(2^{d-1} - w)\Big) \\
&\quad - (2^d - w)d\log_2^2 d + \left(2 + \frac{1}{\zeta}d\log_2 d\right)\log_2 d\log_2(d+1) \\
\overset{(4.26)}{\geq} \quad &2^d\log_2 d - \left(\log_2 d + \frac{1}{d}\right)\zeta 2^{d-1} + \left(2 + \frac{1}{\zeta}d\log_2 d\right)\log_2 d\log_2(d+1)\,.
\end{aligned}$$

By Lemma 4.1.16, this (and, consequently, Equation (4.25)) is non-negative for all $d \geq 3$. This proves Claim 2. □

By Claim 2 and the induction hypothesis, we can find a realization with delay d for $f(\widehat{t^*}, t^{**})$. Split (4.16) hence provides a realization with delay $d+1$ for $f(s,t)$ in case 2.2.2. This concludes the proof. □

Later, in Section 6.1, we will make use of the following observation, which allows us to use a different realization than provided by Lemma 4.1.17 in a special case.

Observation 4.1.18. Consider again case 2.2.1 (b) of the proof of Lemma 4.1.17 in the case that $2^{d-1} < \Lambda_t < 2^d$ and $W(t_{k+1}) > W(t_{k+2})$. Here, the realization used is of the following form:

$$f(s,t) = f\big(s, t' + (t_k, t_{k+2})\big) \wedge f^*\Big(\widehat{t'} + (t_k), t_{k+1}\Big) \qquad (4.27)$$

By the proof, this realization yields a circuit with delay at most $d+1$. As $W(t_{k+1}) = 2^{d-1}$, the sub-circuit for $f^*\Big(\widehat{t'} + (t_k), t_{k+1}\Big)$ will have delay at least d. Together, this implies that this realization yields delay exactly $d+1$.

We must have $m < 8$ as otherwise, we have

$$\begin{aligned}
2^{d-1}\big(\log_2(d+1) - 2\zeta\big) + \log_2(d+1)6(d+1) &\overset{d \geq 3}{\geq} 2^2(2 - 2\zeta) + 2 \cdot 6 \cdot 4 \\
&\overset{\zeta = 1.9}{=} -7.2 + 48 \\
&> 0 \qquad (4.28)
\end{aligned}$$

and thus

$$\Lambda_t \overset{(4.6)}{\leq} \zeta\frac{2^d - w}{\log_2(d+1)} - (d+1)\sum_{i=0}^{m-3} W(t_i) \overset{\substack{w \geq 0, \\ m \geq 8}}{\leq} \zeta\frac{2^d}{\log_2(d+1)} - 6(d+1) \overset{(4.28)}{\leq} 2^{d-1}\,,$$

which contradicts the assumption that $\Lambda_t > 2^{d-1}$. As $m = |t'| + |t''|$, the prefix t' has an odd number of elements and t'' has 3 elements, we thus have $m \in \{4, 6\}$.

In order to realize $f(s,t)$ in this case, we could alternatively apply the alternating split with an even prefix, i.e., Corollary 2.6.18, with prefix $(t_0, \dots, t_k)$:

$$f(s,t) = f\big(s, (t_0, \dots, t_k)\big) \vee f\big(s +\!\!+ (t_0, t_2, \dots, t_{k-1}), (t_{k+1}, t_{k+2})\big)\,. \tag{4.29}$$

The function $f\big(s, (t_0, \dots, t_k)\big)$ can be realized with delay at most d

$$\sum_{i=0}^{k} W(t_i) = W(t) - \Lambda_t \overset{(4.9)}{\leq} \zeta \frac{2^d - w}{(d+1)\log_2(d+1)} - \frac{\Lambda_t}{d+1} \tag{4.30}$$

and thus

$$\begin{aligned}
&\nu(d,w) - \sum_{i=0}^{k} W(t_i) \\
\overset{(4.30)}{\geq}\;& \zeta \frac{2^{d-1} - w}{d \log_2 d} - \left(\zeta \frac{2^d - w}{(d+1)\log_2(d+1)} - \frac{\Lambda_t}{d+1} \right) \\
=\;& \frac{\zeta(2^{d-1} - w)(d+1)\log_2(d+1) - \zeta(2^d - w) d \log_2 d}{d(d+1)\log_2 d \log_2(d+1)} + \frac{\Lambda_t}{d+1} \\
\overset{w<2^{d-1}}{\geq}\;& -\frac{\zeta 2^{d-1} d \log_2 d}{d(d+1)\log_2 d \log_2(d+1)} + \frac{\Lambda_t}{d+1} \\
\overset{\Lambda_t > 2^{d-1}}{\geq}\;& -\frac{\zeta 2^{d-1}}{(d+1)\log_2(d+1)} + \frac{2^{d-1}}{d+1} \\
=\;& \frac{2^{d-1}\big(\log_2(d+1) - \zeta\big)}{(d+1)\log_2(d+1)} \\
\overset{\substack{d+1 \geq 4,\\ \zeta < 2}}{>}\;& 0\,.
\end{aligned}$$

The function $f\big(s +\!\!+ (t_0, t_2, \dots, t_{k-1}), (t_{k+1}, t_{k+2})\big)$ is a symmetric tree which can be realized with at most delay d as $w + W(t) \leq 2^d$, which holds by Lemma 4.1.8 as $d + 1 \geq 4 > 2^\zeta$.

Hence, in this case, the realization (4.29) has delay at most $d+1$ and could be used instead of (4.27) for computing $f(s,t)$. We will make use of this in Section 6.1.

Using the important Lemma 4.1.17, we can now prove all open theorems from this section.

Proof of Theorem 4.1.11. Lemma 4.1.17 proves the theorem in the case that $0 \leq w < 2^{d-1}$, while Lemma 4.1.13 proves it for the remaining case that $2^{d-1} \leq w < 2^d$. □

Proof of Theorem 4.1.6. For $m \leq 2$, Lemma 4.1.9 yields a realization for $f(s,t)$ with delay d. For larger m, we prove the theorem by induction on d. For $d \leq 3$, the required realization can be constructed using Lemma 4.1.10. Now we may assume that the theorem holds for some $d \geq 3$, and prove the inductive step via Theorem 4.1.11. □

Proof of Theorem 4.1.2. As for all input variables t, we have $\Lambda_t \geq 0$, the statement is directly implied by Theorem 4.1.6. □

Algorithm 4.1 states the algorithm for the construction of a formula circuit realizing the extended AND-OR path $f(s,t)$ for given symmetric inputs $s = (s_0, \dots, s_{n-1})$ and alternating inputs $t = (t_0, \dots, t_{m-1})$ arising from the proof of Theorem 4.1.2: In line 5, we compute the minimum $d \in \mathbb{N}$ such that $W(t) \leq \nu(d,w) + \frac{d-1}{d}\Lambda_t$, where $w = W(s)$. Note that in line 5, we have $w < 2^{d-1}$ since otherwise, we would obtain a contradiction to $\Lambda_t \leq W(t)$ since

$$W(t) \leq \zeta \frac{2^{d-1} - w}{d \log_2(d)} + \frac{d-1}{d}\Lambda_t \leq \frac{d-1}{d}\Lambda_t < \Lambda_t\,.$$

Thus, Theorem 4.1.2 provides a circuit realizing $f(s,t)$ with delay d. Lemma 4.1.9 computes this realization for $m \leq 2$ (see line 4), and Lemma 4.1.10 for $d \leq 3$ (see line 7). For $d \geq 4$ and $m \geq 3$, Lemma 4.1.13 (see line 10) and Lemma 4.1.17 (see lines 12 to 29) construct the circuit using recursion. Here, sometimes the dual of the recursively computed circuit is needed, which can be obtained easily by exchanging all AND and OR gates (see Theorem 2.1.31). Altogether, Algorithm 4.1 computes a realization with delay d by Theorem 4.1.2.

Theorem 4.1.19. *Consider $m,n \in \mathbb{N}$ with $m \geq 3$. Given symmetric inputs $s = (s_0, \dots, s_{n-1})$ and alternating inputs $t = (t_0, \dots, t_{m-1})$ and with arrival times $a(s_0), \dots, a(s_{n-1}), a(t_0), \dots, a(t_{m-1}) \in \mathbb{N}$, Algorithm 4.1 computes a circuit realizing the extended* AND-OR *path $f(s,t)$ with delay at most d, where $d \in \mathbb{N}$ is chosen minimum with*

$$W(t) \leq \nu(d, W(s)) + \frac{d-1}{d}\Lambda_t\,. \qquad \square$$

4.2 Delay Analysis

Based on Theorem 4.1.19, we directly show in Proposition 4.2.3 that there is a circuit realizing the AND-OR path $g(t)$ with delay at most $\log_2 W(t) + \log_2\log_2 W(t) + \log_2\log_2\log_2 W(t) + 4.8$. Afterwards, we will prove a stronger result: By modifying the instance, we can reduce the dependency on $W(t)$. The modification is based on two ideas: First, we can round up small arrival times to a common value without losing too much regarding the maximum delay. Secondly, shifting all arrival times by some number does not change the problem. These two modifications allow us to reduce the problem to instances with a total weight of at most $2m$. This leads to a delay bound of $\log_2 W(t) + \log_2\log_2 m + \log_2\log_2\log_2 m + 7.1$ proven in Remark 4.2.5. In a further step, we make use of the fact that for a small number m of inputs, the delay bound of the circuits constructed in Held and Spirkl [HS17b] is better than ours. By this, we obtain our final delay bound of $\log_2 W(t) + \log_2\log_2 m + \log_2\log_2\log_2 m + 4.3$ in Theorem 4.2.4.

The next two lemmas are a common technical preparation for these results.

Lemma 4.2.1. *The function $\kappa : x \mapsto \frac{2^{x-1}}{x \log_2 x}$ is strictly monotonely increasing for all $x \geq 3$.*

Proof. To prove the statement, we compute the derivative of κ:

$$\begin{aligned}\frac{d}{dx}\kappa(x) &= \frac{\ln(2)2^{x-1}x\log_2 x - 2^{x-1}\Big(\log_2 x + x\frac{1}{\ln(2)x}\Big)}{x^2\log_2^2 x} \\ &= 2^{x-1}\frac{\log_2 x\big(\ln(2)x - 1\big) - \frac{1}{\ln(2)}}{x^2 \log_2^2 x}\end{aligned}$$

Algorithm 4.1: Delay optimization for extended AND-OR paths

```
Input: Inputs $s = (s_0, \dots, s_{n-1})$ and $t = (t_0, \dots, t_{m-1})$,
       arrival times $a(s_0), \dots, a(s_{n-1}), a(t_0), \dots, a(t_{m-1}) \in \mathbb{N}$.
Output: Circuit $C(s,t)$ computing $f(s,t)$.
1  Set $w := W(s)$.
2  if $m \leq 2$ then
3      Construct $C(s,t)$ via Huffman coding [Huf52] (Theorem 2.3.21).
4      return $C(s,t)$
5  Choose $d \in \mathbb{N}$ minimum with $W(t) \leq \nu(d,w) + \frac{d-1}{d}\Lambda_t$.   // Hence, $w < 2^{d-1}$.
6  else if $d \leq 3$ then                                    // Hence, $d = 3, m = 3, n = 0$.
7      return $C(s,t) := t_0 \wedge (t_1 \vee t_2)$
8  else if $w \geq 2^{d-2}$ then
9      Recursively compute $C(s,())$ and $C((),t)$.
10     return $C(s,t) := C(s,()) \wedge C((),t)$
11 else
12     if $W(t_0) > \nu(d-1,w)$ then
13         Recursively compute $C(s,(t_0))$ and $C((),(t_1,\dots,t_{m-1}))$.
14         return $C(s,t) = C(s,(t_0)) \wedge \Big(C\big((),(t_1,t_2,\dots,t_{m-1})\big)\Big)^*$
15     else
16         Choose a maximum odd-length prefix $t'$ of $t$ with $W(t') \leq \nu(d-1,w)$.
17         Set $t'' := t \backslash t'$.
18         if $|t''| = 3$ and $\Lambda_t > 2^{d-2}$ and $W(t_{m-1}) < W(t_{m-2})$ then
19             Recursively compute $C(s,t) := C(s,(t_0,\dots,t_{m-3},t_{m-1},t_{m-2}))$.
20             return $C(s,t)$.
21         Set $\tilde{t} := t' \mathbin{+\!\!+} (t''_0, t''_1)$.
22         Set $t^* := \begin{cases} t' & \text{if } |t''| \leq 2 \text{ or } W\left(\tilde{t}\right) > \nu(d-1,w) + \frac{d-2}{d-1}\Lambda_{\tilde{t}}\,, \\ \tilde{t} & \text{otherwise}\,. \end{cases}$
23         Set $t^{**} := t \backslash t^*$.
24         Recursively compute $C(s,t^*)$.
25         if $|t''| = 3$ and $t^* = t'$ then
26             Compute $C^*\left(\widehat{t'}, t''\right) = \left(\widehat{t'} \vee t''_0\right) \vee \left(t''_1 \wedge t''_2\right)$ directly.
27         else
28             Recursively compute $C^*\left(\widehat{t^*}, t^{**}\right) := \left(C\left(\widehat{t^*}, t^{**}\right)\right)^*$.
29         return $C(s,t) := C(s,t^*) \wedge C^*\left(\widehat{t^*}, t^{**}\right)$
```

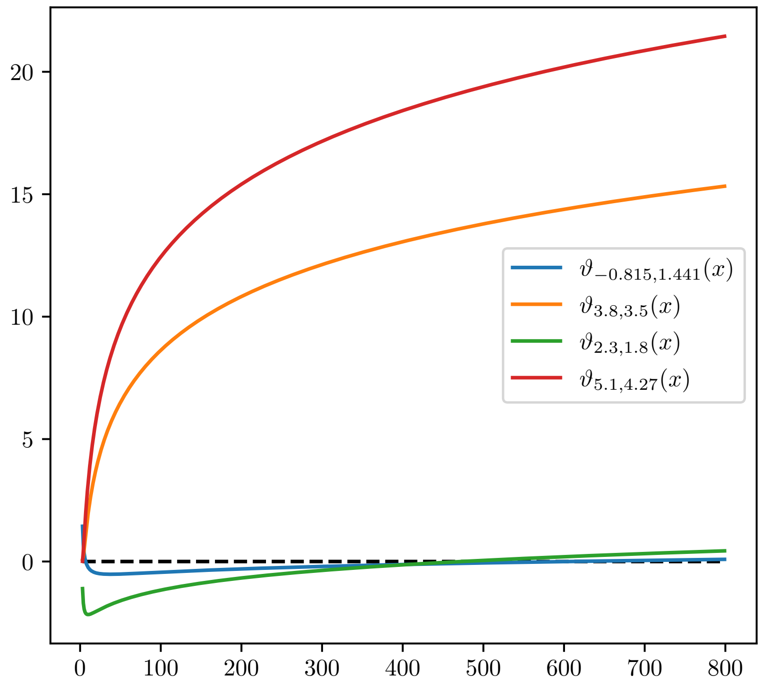


Figure 4.3: Illustration of the various functions from Lemma 4.2.2 in the range $3 \le x \le 800$.

For $x \ge 3$, we have $\ln(2)x - 1 \ge 1$ and $\log_2 x \ge \frac{1}{\ln(2)}$, hence $\frac{d}{dx}\kappa(x)$ is strictly positive and $\kappa(x)$ strictly monotonely increasing for all $x \ge 3$. □

Lemma 4.2.2. *For the parametrized function*

$$\vartheta_{c,\alpha} \colon x \mapsto (\alpha - 1)\log_2 x - \log_2 \log_2 x - \log_2 \log_2 \log_2 x - c\,,$$

the following statements hold:

(i) The function $\vartheta_{-0.815,1.441}$ is negative for all $7 \le x \le 499$.

(ii) The function $\vartheta_{3.8,3.5}$ is positive for all $x \ge 3$.

(iii) The function $\vartheta_{2.3,1.8}$ is positive for all $x \ge 500$.

(iv) The function $\vartheta_{5.1,4.27}$ is positive for all $x \ge 3$.

Figure 4.3 depicts the functions $\vartheta_{c,\alpha}(x)$ in all four cases.

Proof. For the cases (ii) and (iv), we consider $x = 3$ separately. Here, we have

$$\begin{aligned}\vartheta_{c,\alpha}(3) &= (\alpha - 1)\log_2 3 - \log_2 \log_2 3 - \log_2 \log_2 \log_2 3 - c \\ &= \begin{cases} (3.5 - 1)\log_2 3 - \log_2 \log_2 3 - \log_2 \log_2 \log_2 3 - 3.8 & \text{in case (ii)} \\ (4.27 - 1)\log_2 3 - \log_2 \log_2 3 - \log_2 \log_2 \log_2 3 - 5.1 & \text{in case (iv)} \end{cases} \\ &> \begin{cases} 0.08 & \text{in case (ii)} \\ 0.008 & \text{in case (iv)} \end{cases} \\ &> 0\,.\end{aligned}$$

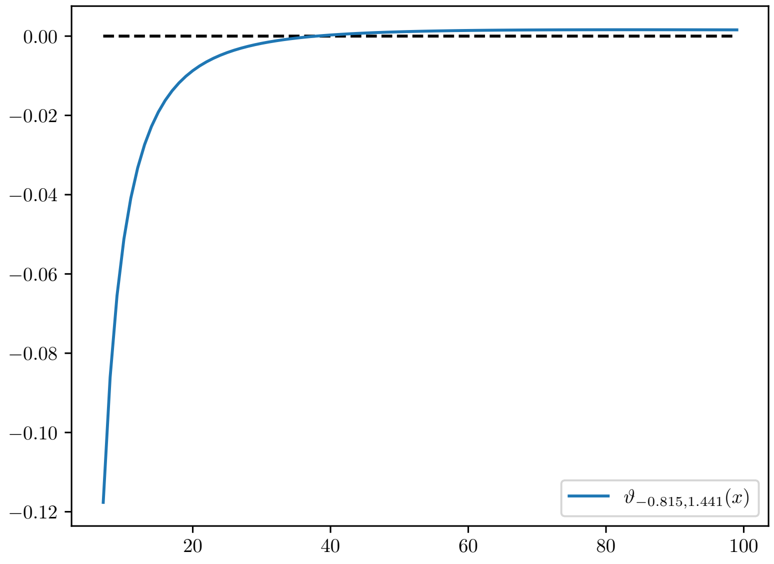


Figure 4.4: Illustration of the function $\frac{d}{dx}\vartheta_{-0.815,1.441}(x)$ from the proof of Lemma 4.2.2 in the range $7 \le x \le 100$.

For all other cases, we compute the derivative of $\vartheta_{c,\alpha}$:

$$\begin{aligned}\frac{d}{dx}\vartheta_{c,\alpha}(x) &= \frac{\alpha-1}{\ln(2)x} - \frac{1}{\ln^2(2)x\log_2 x} - \frac{1}{\ln^3(2)x\log_2 x\log_2\log_2 x}\\ &= \frac{1}{\ln(2)x}\left(\alpha - 1 - \frac{1}{\ln(2)\log_2 x}\left(1+\frac{1}{\ln(2)\log_2\log_2 x}\right)\right)\end{aligned}$$

Figure 4.4 shows the derivative of $\vartheta_{-0.815,1.441}(x)$.

In the cases (ii) and (iv), for $x \ge 4$, we have $1+\frac{1}{\ln(2)\log_2\log_2 x} \le 2.45$ and hence, as $\alpha \ge 3.5$, we have

$$\alpha - 1 - \frac{1}{\ln(2)\log_2 x}\cdot 2.172 > \alpha - 2.8 > 0\,.$$

Thus, $\frac{d}{dx}\vartheta_{c,\alpha}(x)$ is positive for $x \ge 4$ in the cases (ii) and (iv).

In case (iii), we have $x \ge 500$ and $1+\frac{1}{\ln(2)\log_2\log_2 x} \le 1.46$ and hence, as $\alpha = 1.8$, we have

$$\alpha - 1 - \frac{1}{\ln(2)\log_2 x}\cdot 1.46 > \alpha - 1.3 > 0\,.$$

Hence, $\frac{d}{dx}\vartheta_{2.3,1.8}(x)$ is positive.

In case (i), for $7 \le x \le 37$, we have $1+\frac{1}{\ln(2)\log_2\log_2 x} \ge 1.6$ and hence, as $\alpha = 1.441$, we have

$$\alpha - 1 - \frac{1}{\ln(2)\log_2 x}\cdot 1.6 < \alpha - 1.443 < 0\,;$$

and for $38 \le x \le 499$, we have $1+\frac{1}{\ln(2)\log_2\log_2 x} \le 1.604$ and hence, as $\alpha = 1.441$, we have

$$\alpha - 1 - \frac{1}{\ln(2)\log_2 x}\cdot 1.61 > \alpha - 1.441 = 0\,.$$

Hence, $\frac{d}{dx}\vartheta_{-0.815,1.441}(x)$ is negative for $7 \leq x \leq 37$ and positive for $38 \leq x \leq 499$.

From this, we can conclude the proof:

$$\begin{aligned}
\vartheta_{-0.815,1.441}(x) &\overset{7\leq x\leq 37}{\leq} \vartheta_{-0.815,1.441}(7) < -0.01 < 0\\
\vartheta_{-0.815,1.441}(x) &\overset{38\leq x\leq 499}{\leq} \vartheta_{-0.815,1.441}(4) < -0.05 < 0\\
\vartheta_{3.8,3.5}(x) &\overset{x\geq 4}{\geq} \vartheta_{3.8,3.5}(4) = 0.2 > 0\\
\vartheta_{2.3,1.8}(x) &\overset{x\geq 500}{\geq} \vartheta_{2.3,1.8} > 0.04 > 0\\
\vartheta_{5.1,4.27}(x) &\overset{x\geq 4}{\geq} \vartheta_{5.1,4.27} = 0.44 > 0
\end{aligned}$$ □

We will now bound the delay of the circuits arising from Theorem 4.1.19.

Proposition 4.2.3. *Let $m \in \mathbb{N}$ with $m \geq 3$, Boolean variables $t_0, \dots, t_{m-1}$ and arrival times $a(t_0), \dots, a(t_{m-1}) \in \mathbb{N}$ be given. There is a circuit realizing the* AND-OR *path $g(t)$ with delay at most*

$$\log_2 W(t) + \log_2 \log_2 W(t) + \log_2 \log_2 \log_2 W(t) + 4.8\,.$$

Proof. Let $d := \lfloor \log_2 W(t) + \log_2 \log_2 W(t) + \log_2 \log_2 \log_2 W(t) + 4.8 \rfloor$. By Theorem 4.1.19, it suffices to show that $W(t) \leq \nu(d, w)$, i.e., that

$$W(t) \leq \zeta \frac{2^{d-1}}{d \log_2 d}\,.$$

By Lemma 4.2.1, the right-hand side function is strictly increasing for $d \geq 3$, so it suffices to show that

$$W(t) \leq \zeta \frac{2^{d'-1}}{d' \log_2 d'} \tag{4.31}$$

for $d' := \log_2 W(t) + \log_2 \log_2 W(t) + \log_2 \log_2 \log_2 W(t) + 3.8$. For $W(t) \geq m \geq 3$, we have

$$d' \leq 3.5 \log_2 W(t)$$

by Lemma 4.2.2, case (ii). Thus, Equation *(4.31)* is implied by

$$W(t) \leq \zeta \frac{2^{d'-1}}{3.5 \log_2 W(t) \log_2(3.5 \log_2 W(t))}\,,$$

which is equivalent to

$$3.5 W(t) \log_2 W(t) \log_2(3.5 \log_2 W(t)) \leq \zeta 2^{\log_2 W(t) + \log_2 \log_2 W(t) + \log_2 \log_2 \log_2 W(t) + 2.8}\,,$$

and hence to

$$3.5 \log_2 \log_2 W(t) + 3.5 \log_2(3.5) \leq \zeta 2^{2.8} \log_2 \log_2 W(t)\,,$$

which is true since $W(t) \geq 3$ and $\zeta \geq 1.9$. □

By transforming the instance and using different algorithms for small instances, we can improve this delay bound as shown in the following theorem.

Theorem 4.2.4. *Let $m \in \mathbb{N}$ with $m \geq 3$, Boolean variables $t_0, \dots, t_{m-1}$ and arrival times $a(t_0), \dots, a(t_{m-1}) \in \mathbb{N}$ be given. There is a circuit realizing the* And-Or *path $g(t)$ with delay at most*

$$\log_2 W(t) + \log_2 \log_2 m + \log_2 \log_2 \log_2 m + 4.3\,.$$

Proof. We compute auxiliary arrival times $\tilde{a} : \{t_0, \dots, t_{m-1}\} \to \mathbb{N}$ by setting

$$\tilde{a}(t_i) := \max\Big\{0, a(t_i) - \big\lceil \log_2 W(t) - \log_2 m \big\rceil\Big\}$$

for all $i \in \{0, \dots, m-1\}$. Let $\widetilde{W} := W(t; \tilde{a})$. We partition the input indices $\{0, \dots, m-1\}$ into

$$I_1 := \big\{i \in \{0, \dots, m-1\} : \tilde{a}(t_i) = 0\big\} \quad \text{and} \quad I_2 := \{0, \dots, m-1\} \setminus I_1\,.$$

Then, we have

$$\begin{aligned}
\widetilde{W} &= \sum_{i \in I_1} 2^{\tilde{a}(t_i)} + \sum_{i \in I_2} 2^{\tilde{a}(t_i)} \\
&= |I_1| + \sum_{i \in I_1} 2^{a(t_i) - \lceil \log_2 W(t) - \log_2 m \rceil} \\
&\leq m + 2^{-\lceil \log_2 W(t) - \log_2 m \rceil} \sum_{i \in I_2} 2^{a(t_i)} \\
&\leq m + \frac{2^{\log_2 m}}{2^{\log_2 W(t)}} W(t) \\
&= 2m\,. \qquad (4.32)
\end{aligned}$$

Note that this bound is best possible as in the case that $a(t_0) = \ldots = a(t_{m-1}) = 0$ and $a(t_{m-1})$ very large in comparison to m, the sum $\widetilde{W}$ gets arbitrarily close to $2m$.

Let $c := 4.3$ be the additive constant in the delay bound of this theorem. Define

$$\tilde{d} := \lfloor \log_2 m + \log_2 \log_2 m + \log_2 \log_2 \log_2 m + c - 1 \rfloor\,.$$

We shall now first construct a circuit C with delay $\tilde{d}$ for inputs with arrival times $\tilde{a}$ in the following claim; from this, we shall later deduce the theorem.

Claim. There is a circuit C realizing the And-Or path $g(t)$ with $\text{delay}(C; \tilde{a}) \leq \tilde{d}$.

Proof of claim: Let $M := 500$. For $m < M$, we give a realization for $g(t)$ without using our results. Note that $\tilde{a}(t_i) = \log_2 W(t_i; \tilde{a}) \leq \log_2 \widetilde{W} \leq \log_2 m + 1$ for all $i = 0, \dots, m-1$. Hence, the standard realization C^S_m for m inputs has delay at most $\big\lfloor \max_{i=0,\dots,m-1} \tilde{a}(t_i) + m - 1 \big\rfloor \leq \lfloor m + \log_2 m \rfloor$. Hence, we have

$$\begin{aligned}
\text{delay}(C^S_3) &\leq \lfloor 3 + \log_2 3 \rfloor = 4 = \lfloor \log_2 3 + 3.3 \rfloor && < \tilde{d} \\
\text{delay}(C^S_4) &\leq \lfloor 4 + \log_2 4 \rfloor = 6 = \lfloor \log_2 4 + \log_2 \log_2 4 + 3.3 \rfloor && < \tilde{d} \\
\text{delay}(C^S_5) &\leq \lfloor 5 + \log_2 5 \rfloor = 7 = \lfloor \log_2 5 + \log_2 \log_2 5 + \log_2 \log_2 \log_2 5 + 3.3 \rfloor && = \tilde{d}\,,
\end{aligned}$$

implying that for $3 \leq m \leq 5$, choosing $C := C^S_m$ solves the claim.

If $7 \leq m < M$, we have

$$\begin{aligned}
\Big\lfloor 1.441 \log_2 \widetilde{W} + 2.674 \Big\rfloor &\overset{(4.32)}{\leq} \lfloor 1.441 \log_2(2m) + 2.674 \rfloor \\
&= \lfloor 1.441 \log_2 m + 4.115 \rfloor \\
&\overset{\text{Lem. 4.2.2}}{\leq} \lfloor \log_2 m + \log_2 \log_2 m + \log_2 \log_2 \log_2 m + c - 1 \rfloor \\
&= \tilde{d}\,.
\end{aligned}$$

For $m = 6$, we also have $\lfloor 1.441 \log_2 m + 4.115 \rfloor = 7 = \tilde{d}$. Since by Theorem 2.6.28, the AND-OR path optimization method by Held and Spirkl [HS17b] computes a circuit with delay at most $\left\lfloor 1.441 \log_2 \widetilde{W} + 2.674 \right\rfloor$, this proves the claim for $6 \leq m < M$.

Hence, assume $m \geq M$. For proving the claim, by Theorem 4.1.19, it is sufficient to show

$$2m \leq \zeta \frac{2^{\tilde{d}-1}}{\tilde{d} \log_2 \tilde{d}}. \tag{4.33}$$

Recall that the mapping $x \mapsto \frac{2^{x-1}}{x \log_2 x}$ is strictly increasing for $x \geq 3$ by Lemma 4.2.1. For $\tilde{\tilde{d}} := \log_2 m + \log_2 \log_2 m + \log_2 \log_2 \log_2 m + c - 2$, we have $\tilde{d} \geq \tilde{\tilde{d}} \overset{m \geq M = 500}{>} 3$. Moreover, for $m \geq M$, we have

$$\log_2 m + \log_2 \log_2 m + \log_2 \log_2 \log_2 m + c - 2 \leq 1.8 \log_2 m \tag{4.34}$$

by Lemma 4.2.2, case (iii), Thus, we have

$$\begin{aligned}
\zeta \frac{2^{\tilde{d}-1}}{\tilde{d} \log_2 \tilde{d}} &\overset{\substack{\tilde{d} \geq \tilde{\tilde{d}} > 3, \\ \text{Lem. 4.2.1}}}{\geq} \zeta \frac{2^{\tilde{\tilde{d}}-1}}{\tilde{\tilde{d}} \log_2 \tilde{\tilde{d}}} \\
&\overset{(4.34)}{\geq} \zeta \frac{2^{\tilde{\tilde{d}}-1}}{1.8 \log_2 m \log_2(1.8 \log_2 m)} \\
&\overset{\text{def. } \tilde{\tilde{d}}}{=} \zeta \frac{m \log_2 \log_2 m 2^{c-3}}{1.8 \log_2(1.8 \log_2 m)}.
\end{aligned}$$

Equation (4.33) is hence valid if

$$1.8 \log_2 \log_2 m + 1.8 \log_2(1.8) \leq \zeta 2^{c-4} \log_2 \log_2 m. \tag{4.35}$$

Since $\zeta 2^{c-4} > 2.3$, this statement is fulfilled for large enough m, so it suffices to prove it for $m = M$. Here, we have

$$1.8 \log_2 \log_2 M + 1.8 \log_2(1.8) < 7.3 < 7.4 < \zeta 2^{c-4} \log_2 \log_2 m.$$

This proves Equation (4.33) and hence the claim. □

Since we have $a(t_i) \leq \tilde{a}(t_i) + \lceil \log_2 W(t) - \log_2 m \rceil$ for all $i \in \{0, \ldots, m-1\}$, the circuit C fulfills

$$\begin{aligned}
&\text{delay}(C; a) \\
\leq\ & \tilde{d} + \lceil \log_2 W(t) - \log_2 m \rceil \\
=\ & \lfloor \log_2 m + \log_2 \log_2 m + \log_2 \log_2 \log_2 m + c - 1 \rfloor + \lceil \log_2 W(t) - \log_2 m \rceil \\
\leq\ & \log_2 W(t) + \log_2 \log_2 m + \log_2 \log_2 \log_2 m + c.
\end{aligned}$$

□

Remark 4.2.5. In the proof of the previous theorem, we apply method [HS17b] for small instances. Without this trick, we would obtain a delay bound of

$$\log_2 W(t) + \log_2 \log_2 m + \log_2 \log_2 \log_2 m + 7.1$$

by altering the proof of Theorem 4.2.4 as follows: Now, we have $c = 7.1$ and $M := 3$. For these values, the constant 1.8 in Equation (4.34) increases to 4.27 by Lemma 4.2.2, case (iv), and Equation (4.35) still holds.

Remark 4.2.6. For sufficiently large values of m, the delay bound in the previous theorem can be improved slightly to

$$\log_2 W(t) + \log_2 \log_2 m + \log_2 \log_2 \log_2 m + 3.1 + \varepsilon$$

for any constant $\varepsilon > 0$: Note that the factor 1.8 in inequality (4.34) can be decreased to a value arbitrarily close to 1 if m is sufficiently large. Thus, we may choose c such that the factor $\zeta 2^{c-4}$ in inequality (*4.35*) becomes arbitrarily close to 1 for large values of m. This leads to the stated delay bound.

4.3 Analysis of Algorithm and Circuit

We shall now analyze the delay optimization algorithm presented in Theorem 4.2.4, which has Algorithm 4.1 (page 119) as its core routine, and the circuits arising from this algorithm.

Our main objective when designing good circuits for And-Or paths is delay. Still, there are other metrics to be regarded during circuit construction such as the size, i.e., the total number of gates used in the circuit, and maximum fanout, i.e., the maximum number of successors of any input or gate.

Proposition 4.3.1. *The circuit C computed in Proposition 4.3.3 fulfills*

$$\text{size}(C) \leq m(\log_2 m + \log_2 \log_2 m + \log_2 \log_2 \log_2 m + 3.3) - 1$$

and

$$\text{fanout}(C) \leq \log_2 m + \log_2 \log_2 m + \log_2 \log_2 \log_2 m + 3.3\,.$$

Proof. We follow the proof of Proposition 4.3.3. For $m \leq 5$, we construct the standard realization with size $m-2$ and maximum fanout 1, and for $6 \leq m < M$, we apply the method from Held and Spirkl [HS17b] with size at most $1.5m$ and maximum fanout 2 as described in Theorem 2.6.28. Both types of circuits fulfill the required bounds.

Now assume that $m > M$, where C arises from applying Algorithm 4.1 (page 119) to modified arrival times $\tilde{a}$ as defined in the proof of Theorem 4.2.4. In order to prove the fanout bound, we show the following claim.

Claim. In the circuit computed by Algorithm 4.1 (page 119), each gate has fanout exactly 1, each input in s has fanout exactly 1 and each input in t has fanout at most d, where d is as computed in line 5 of Algorithm 4.1.

Proof of claim: Note that each gate constructed has fanout 1 and that a fanout higher than 1 occurs at the inputs only. We prove the bound on the maximum fanout of the inputs by induction on d.

Note that in the realizations computed by Lemmas 4.1.9 and 4.1.10 which are used in lines 3 and 7, respectively, each input has fanout 1. In the realization $C(s,t) := C(s,()) \wedge C((),t)$ provided in line 10, the inputs of the two recursive constructions are disjoint. Hence, by induction hypothesis, we have fanout 1 for all inputs in s and fanout $d-1 < d$ for all inputs in t.

By Equation (4.27), using the notation from the proof of Lemma 4.1.17 and Algorithm 4.1, the circuit computed in line 19 is $C(s,t) = C\big(s, t' +\!\!+ (t_k, t_{k+2})\big) \wedge C^*\Big(\widehat{t'} +\!\!+ (t_k), t_{k+1}\Big)$. As $C^*\Big(\widehat{t'} +\!\!+ (t_k), t_{k+1}\Big)$ is a symmetric tree, all its inputs will have fanout 1. The circuit $C\big(s, t' +\!\!+ (t_k, t_{k+2})\big)$ will be computed by recursion with fanout at most 1 for inputs in s and fanout at most $d-1$ for the inputs in t. Summing up, this yields the required fanout bounds.

In the realizations in lines 14 and 29, we perform the split $C(s,t) = C(s,t^*) \wedge C^*\left(\widehat{t^*}, t^{**}\right)$, where $t^* = (t_0)$ in line 14. By induction hypothesis, the circuit $C(s,t^*)$ has fanout at most 1 for inputs in s and fanout at most $d-1$ for inputs in t^*. Since inputs of s do not occur in $C^*\left(\widehat{t^*}, t^{**}\right)$, it remains to show that inputs of t have fanout at most d in $C(s,t)$.

In line 14, this holds as no input of t occurs in both sub-circuits and by induction hypothesis, the inputs of $C^*\left(\widehat{t^*}, t^{**}\right)$ have depth at most $d-1 < d$. In the realization of $C^*\left(\widehat{t^*}, t^{**}\right)$ in 26, each input of t has fanout at most 1 in $C^*\left(\widehat{t^*}, t^{**}\right)$, which proves the claimed fanout bounds. Otherwise, we construct $C^*\left(\widehat{t^*}, t^{**}\right)$ recursively in line 28 and we inductively can assume that the inputs of $\widehat{t^*}$ have fanout at most 1 and the inputs of t^{**} fanout at most $d-1$ in this realization. Together with the recursive fanout bounds for $C(s,t^*)$, this shows the claimed fanout bounds for $C(s,t)$. □

The claim implies

$$\operatorname{fanout}(C) \leq \operatorname{delay}(C;\tilde{a}) \overset{\text{Thm. 4.2.4}}{\leq} \log_2 m + \log_2\log_2 m + \log_2\log_2\log_2 m + 3.3\,,$$

which shows the proposed bound on the maximum fanout.

As for $m \geq M$, by the claim, the constructed circuit is a formula circuit, the fanout bound together with Observation 2.3.6 implies that

$$\begin{aligned}\operatorname{size}(C) &= \sum_{v\in\mathcal{I}} \operatorname{fanout}(v) - 1\\ &\leq m(\log_2 m + \log_2\log_2 m + \log_2\log_2\log_2 m + 3.3) - 1\,.\end{aligned}$$ □

The following lemma is a preparation for the running time analysis in Proposition 4.3.3.

Lemma 4.3.2. *Let alternating inputs $t = (t_0, \ldots, t_{m-1})$ and symmetric inputs $s = (s_0, \ldots, s_{n-1})$ with arrival times $a(t_0), \ldots, a(t_{m-1}), a(s_0), \ldots, a(s_{n-1}) \in \mathbb{N}$ be given. For $w = W(s)$, consider the minimum value $d \in \mathbb{N}$ such that $W(t) \leq \nu(d,w) + \frac{d-1}{d}\Lambda_t$. Then, we have $d \in \mathcal{O}(\log_2(W'))$, where $W' = W(t) + w$.*

Proof. Consider the modified instance with no symmetric inputs and alternating inputs $t_0, \ldots, t_{m-1}, s_0, \ldots, s_{n-1}$, with arrival times as for the original instance. Choose $d' \in \mathbb{N}$ minimum with $W' \leq \nu(d',0) + \frac{d'-1}{d'}\Lambda_t$. Applying Theorem 4.2.4 to this modified instance yields $d' \leq \log_2 W' + \log_2\log_2(m+n) + \log_2\log_2\log_2(m+n) + 4.3$. But we have

$$\begin{aligned}&\min\left\{d' : W' \leq \nu(d',0) + \frac{d'-1}{d'}\Lambda_t\right\}\\ =\ &\min\left\{d' : W(t) + w \leq \zeta\frac{2^{d-1}}{d\log_2 d} + \frac{d'-1}{d'}\Lambda_t\right\}\\ \geq\ &\min\left\{d : W(t) + \frac{w}{d\log_2 d} \leq \zeta\frac{2^{d-1}}{d\log_2 d} + \frac{d-1}{d}\Lambda_t\right\}\\ =\ &\min\left\{d : W(t) \leq \nu(d,w) + \frac{d-1}{d}\Lambda_t\right\},\end{aligned}$$

which implies that $d \leq d' \in \mathcal{O}(\log_2(W'))$. □

Proposition 4.3.3. *There is an algorithm that computes the circuit in Theorem 4.2.4 for given $m \geq 3$ in time $\mathcal{O}(m \log_2^2 m)$.*

Proof. As main subroutine, we will use Algorithm 4.1 (page 119).

Claim. Let input variables $s = (s_1, \dots, s_{n-1})$ and $t = (t_0, \dots, t_{m-1})$ with arrival times $a : \{t_0, \dots, t_{m-1}, s_0, \dots, s_{n-1}\} \to \mathbb{N}$ be given. The number of computation steps of Algorithm 4.1 to compute the circuit $C(s,t)$ realizing $f(s,t)$ is bounded by

$$\mathcal{O}\Big(\operatorname{size}(C) \log_2\big(\operatorname{size}(C)\big) + m\big(\log_2 \log_2(W') + \log_2 m\big)\Big),$$

where $W' = W(t) + W(s)$.

Proof of claim: For proving the bound on the number of computation steps of Algorithm 4.1, we bound the number of steps to construct symmetric trees in line 3, the number of recursive calls to Algorithm 4.1 except for those leading to $m \leq 2$, and the number of steps needed for a single call excluding the recursive calls (i.e., lines 9, 13, 19, 24 and 28) and symmetric tree constructions (i.e., line 3).

First, we derive the total number of computation steps needed to construct symmetric trees in line 3 during the algorithm. The total number of inputs involved in symmetric trees (counting inputs multiply if they occur in multiple symmetric trees) is at most $2 \operatorname{size}(C)$ as C is a binary circuit. The number of computation steps of a single application of Huffman coding to r inputs is $\mathcal{O}(r \log_2 r)$ by Theorem 2.3.21. As this is super-linear, the total number of steps to perform Huffman coding for any symmetric tree can be bounded by $\mathcal{O}\Big(\operatorname{size}(C) \log_2\big(\operatorname{size}(C)\big)\Big)$.

Secondly, we bound the number of recursive calls to Algorithm 4.1 that do not lead to $m \leq 2$. These calls are performed in lines 9, 13, 19, 24 and 28.

The calls in lines 13, 24 and 28 all result from alternating splits, i.e., a split of the form

$$f(s,t) = f(s,t^*) \wedge f^*(\widehat{t^*}, t^{**}) \tag{4.36}$$

for some subset t^* of t. Note that the inputs of t^* might not be consecutive in the overall instance. Furthermore, by Observation 4.1.18, the recursive call in line 19 results in an alternating split in the next recursive call; and the recursive computation of $C((),t)$ in line 9 either results in an alternating split in the next recursive call, or in a call of line 19. Hence, the total number of calls to lines 9, 13, 24 and 28 is linear in the number of alternating splits. By induction on d, one can observe that the function mapping an alternating split as in Equation (4.36) occurring during the algorithm to $(t^{**})_0$ is injective. This bounds the number of recursive calls of Algorithm 4.1 to lines 9, 13, 24 and 28 by $\mathcal{O}(m)$.

Thirdly, we estimate the number of steps of a single call of Algorithm 4.1.

The only case when the inputs of t are not consecutive is when inputs of t were swapped in line 19. But by Observation 4.1.18, this only occurs for constant m. So apart from computing symmetric trees (as n might not be a constant), the number of computation steps of a single call of Algorithm 4.1 is constant in this case.

Hence, we may now assume that t is a consecutive subset of the inputs.

In line 5, we compute d as the minimum integer satisfying $W(t) \leq \nu(d,w) + \frac{d-1}{d}\Lambda_t$. By Lemma 4.3.2, we have $d \in \mathcal{O}(\log_2(W'))$, so d can be computed by binary search in $\mathcal{O}(\log_2 \log_2(W'))$ steps.

For estimating the running time for the prefix computation in line 16, we use that t is consecutive by assumption. If we precompute the weight for each consecutive prefix of the overall alternating inputs of Algorithm 4.1, computing the prefix t'

in line 16 (or finding out that no such prefix exists) requires $\mathcal{O}(\log_2 m)$ steps using binary search. Here, the weight of a consecutive tuple of alternating inputs $(t_i, \dots, t_j)$ can be computed in constant time by $W(t_i, \dots, t_j) = W(t_0, \dots, t_j) - W(t_0, \dots, t_j)$. The precomputation requires $\mathcal{O}(m)$ steps.

Apart from this, there are only constantly many steps in each recursive call.

Hence, the number of steps needed for each recursive call of Algorithm 4.1, excluding lines 3, 9, 13, 24 and 28, is at most $\mathcal{O}(\log_2 \log_2(W') + \log_2 m)$. Since there are at most $\mathcal{O}(m)$ recursive calls and $\text{size}(C) \log_2\big(\text{size}(C)\big)$ steps for performing Huffman coding, we have at most $\mathcal{O}\Big(\text{size}(C) \log_2\big(\text{size}(C)\big) + m\big(\log_2 \log_2(W') + \log_2 m\big)\Big)$ steps in total, which finishes the proof of the claim. □

Now we can prove the proposition. We follow the proof of Theorem 4.2.4, also using its notation. We compute the modified instance with arrival times $\tilde{a}$ and weight $\widetilde{W}$ in linear time. For $m \leq 5$, we construct the standard And-Or path realization in linear time, and for $6 \leq m < M$, we construct the circuit described in [HS17b] in time $\mathcal{O}(m \log_2 m)$ (see also Theorem 2.6.28). For $m \geq M$, we call Algorithm 4.1 with the modified arrival times $\tilde{a}$. Since $\widetilde{W} \in \mathcal{O}(m)$, the sizes of all numbers occurring in the algorithm are polynomial in m. By Proposition 4.3.1, we have $\text{size}(C) \in \mathcal{O}(m \log_2 m)$. Applying the claim with $s = ()$, hence $n = 0$, and $W' = \widetilde{W} \in \mathcal{O}(m)$, we obtain a total running time of

$$\begin{aligned} & \mathcal{O}\Big(\text{size}(C) \log_2\big(\text{size}(C)\big) + m\big(\log_2 \log_2(W') + \log_2 m\big)\Big) \\ = \; & \mathcal{O}\Big(m \log_2^2 m + m(\log_2 \log_2 m + \log_2 m)\Big) \\ = \; & \mathcal{O}\Big(m \log_2^2 m\Big) \,. \end{aligned}$$ □

Combining the results from Theorem 4.2.4, Proposition 4.3.3 and Proposition 4.3.1, we conclude Chapter 4 with the following theorem.

Theorem 4.3.4. *Given inputs $t = (t_0, \dots, t_{m-1})$ with $m \geq 3$ and with arrival times $a(t_0), \dots, a(t_{m-1}) \in \mathbb{N}$ and $W := \sum_{i=0}^{n-1} 2^{a(t_i)}$ as in Definition 2.3.16, we can construct an* And-Or *path circuit C on t with*

$$\begin{aligned} \text{delay}(C) &\leq \log_2 W + \log_2 \log_2 m + \log_2 \log_2 \log_2 m + 4.3 \,, \\ \text{size}(C) &\leq m(\log_2 m + \log_2 \log_2 m + \log_2 \log_2 \log_2 m + 3.3) - 1 \,, \end{aligned}$$

and

$$\text{fanout}(C) \leq \log_2 m + \log_2 \log_2 m + \log_2 \log_2 \log_2 m + 3.3$$

in running time $\mathcal{O}(m \log_2^2 m)$. □

Remark 4.3.5. For given $m \geq 3$, with the additional use of buffers, the circuit C constructed in Theorem 4.3.4 can be transformed into a logically equivalent circuit C' with maximum fanout 2,

$$\text{delay}(C') \leq \log_2 W + 2 \log_2 \log_2 m + \log_2 \log_2 \log_2 m + 5.3$$

and

$$\text{size}(C') \leq 2m(\log_2 m + \log_2 \log_2 m + \log_2 \log_2 \log_2 m + 3.3) - m - 1 \,.$$

To see this, first note that the standard And-Or path circuit and the circuit constructed in [HS17b] which we use for instances with $m < M$ already have a maximum fanout of 2 or even 1, respectively. Secondly, note that the circuit C constructed in Proposition 4.3.3 for $m \geq M$ is a formula circuit by the claim in Proposition 4.3.1, i.e., only inputs have fanout larger than 1. Write

$$f := \log_2 m + \log_2 \log_2 m + \log_2 \log_2 \log_2 m + 3.3$$

for the maximum possible fanout of C. For each input t_i, we can replace the outgoing edges of t_i by a delay-optimum buffer tree with maximum fanout 2 for each buffer (note this can also be computed by Theorem 2.3.21). This increases the size by at most $m(f-1)$ and, since we can assume that $m \geq 500$, the delay by at most $\log_2 f \leq \log_2 \log_2 m + 1$. This yields the stated properties of the transformed circuit.

Chapter 5 Exact Delay Optimization Algorithm

In this chapter, we present an exact algorithm for the Generalized And-Or Path Circuit Optimization Problem. On general instances, our algorithm has a running time of up to $\mathcal{O}(3^m)$, and for the special case of And-Or paths, even only $\mathcal{O}\Big(\big(\sqrt{6}\big)^m\Big)$. For depth optimization of And-Or paths, the running time is further reduced to $\mathcal{O}(m2.02^m)$.

The theoretical running time of the formula enumeration algorithm for depth optimization of And-Or paths by Hegerfeld [Heg18] is $\mathcal{O}\Big(\big(\sqrt{6}\big)^m\Big)$. Hegerfeld computes a depth-optimum formula circuit which has optimum size among all depth-optimum formula circuits C where for each vertex $v \in \mathcal{V}(C)$, the sub-circuit C_v is depth-optimum. We can also compute such a circuit, but besides offer a significantly faster mode where size is ignored and only a delay-optimum circuit is computed.

In contrast to Hegerfeld, in practice, we apply very efficient pruning techniques which drastically reduce the empirical running time. The largest instance Hegerfeld can solve has 29 inputs, while our algorithm with size optimization can solve instances with up to 42 inputs; without size optimization even up to 64 inputs. Our running times on 26 inputs are 2.1 seconds with size optimization and 0.007 seconds without size optimization, while Hegerfeld's running time is 17 hours. Our largest running time without size optimization on any of these instances is less than 3 hours.

From our computations, we deduce the optimum depths of adder circuits on 2^k bits, where $k \leq 13$. As far as we know, we are the first to obtain such a result.

Recall from Corollary 2.3.12 that for every circuit, there is a formula with the same delay, hence also a formula circuit with the same delay. This allows us to focus on formula circuits in this chapter.

In Section 5.1, we develop a generic approach that can extend a circuit optimization algorithm working only for integral arrival times to fractional arrival times. In Section 5.2, we analyze the structure of certain formula circuits for generalized And-Or paths which have minimum delay. From this, in Section 5.3, we derive our exact algorithm, which is refined for the case of depth optimization of And-Or paths in Section 5.4. Practical speed-ups are presented in Section 5.5. In Section 5.6, we show computational results, i.e., our practical running times and the computed optimum adder depths.

5.1 From Integral to Fractional Arrival Times

In Proposition 5.1.3, we present a generic approach to extend a given circuit construction algorithm which works on integral arrival times only to fractional arrival times while only losing a running time factor of $\mathcal{O}(n)$. In Theorem 5.1.5, we improve the additional running time factor to $\mathcal{O}(\log_2 n)$ using a specific binary search.

Definition 5.1.1. Consider a Boolean function $f \colon \{0,1\}^n \to \{0,1\}$ with input variables $x_0, \dots, x_{n-1}$ and let S denote a non-empty set of circuits realizing f. We call a circuit $C \in S$ an ***S*-optimum circuit** for f with respect to arrival times $a(x_0), \dots, a(x_{n-1}) \in \mathbb{R}$ if $\operatorname{delay}(C; a) \leq \operatorname{delay}(C'; a)$ for all $C' \in S$. We say that an algorithm A is ***S*-optimum** if it always computes an S-optimum circuit for f.

For example, let f be a generalized AND-OR path $f = h(t; \Gamma)$ with inputs $t = (t_0, \dots, t_{m-1})$ and gate types $\Gamma = (\circ_0, \dots, \circ_{m-2})$ and S be the set of all formula circuits for f. Then, in Algorithm 5.1, we will see an S-optimum algorithm. However, in our practical implementation of Algorithm 5.1 which applies the speed-up techniques from Section 5.5, we will assume that the arrival times are integral. Hence, in this section, we will show how to extend an S-optimum algorithm A which is defined only for integral arrival times to fractional arrival times. For this, we need the following lemma.

Lemma 5.1.2. *Given a Boolean function f with input variables $x_0, \dots, x_{n-1}$, a non-empty set S of circuits for f, the following statements are fulfilled:*

(i) Consider arrival times a and a number $\alpha \in \mathbb{R}$. Let arrival times $\tilde{a}$ be given by $\tilde{a}(x_i) = a(x_i) + \alpha$ for each $i \in \{0, \dots, n-1\}$. Then, a circuit $C \in S$ is S-optimum for arrival times a with delay $d \in \mathbb{R}$ if and only if it is S-optimum for arrival times $\tilde{a}$ with delay $d + \alpha \in \mathbb{R}$.

(ii) Consider arrival times $a, \tilde{a}$ with $a(x_i) \leq \tilde{a}(x_i)$ for all $i \in \{0, \dots, n-1\}$ and two circuits $C, \widetilde{C} \in S$. Assume that C is S-optimum with respect to arrival times a. Then, we have $\operatorname{delay}(C; a) \leq \operatorname{delay}(\widetilde{C}; \tilde{a})$.

Proof. The first statement holds as for any circuit C, we have $\operatorname{delay}(C; \tilde{a}) = \operatorname{delay}(C; a) + \alpha$.

To see the second statement, note that $\widetilde{C}$ is a circuit for f with delay $\operatorname{delay}(\widetilde{C}; a) \leq \operatorname{delay}(\widetilde{C}; \tilde{a})$ as $a(x_i) \leq \tilde{a}(x_i)$ for each $i \in \{0, \dots, n-1\}$. As C is S-optimum for arrival times a and $\widetilde{C} \in S$, we have $\operatorname{delay}(C; a) \leq \operatorname{delay}(\widetilde{C}; a)$. Together, this yields the second statement. □

Proposition 5.1.3. *Consider a Boolean function $f \colon \{0,1\}^n \to \{0,1\}$ with input variables $x_0, \dots, x_{n-1}$ and a non-empty set S of circuits for f. Assume that algorithm A is an S-optimum algorithm with running time $\mathcal{O}(r(n))$ which is only defined for integral input arrival times. Then, there is an S-optimum algorithm A' for arbitrary fractional input arrival times with running time $\mathcal{O}(r(n)n)$. We call A' the **linear-search extension** of algorithm A.*

Proof. By Lemma 5.1.2, (i), we may assume that all arrival times are non-negative.

Given a fractional number $b \in \mathbb{R}$, let $\operatorname{fp}(b) := b - \lfloor b \rfloor \in [0,1)$ denote the fractional part of b. Let $F := \big\{ \operatorname{fp}(a(x_i)) : i \in \{0, \dots, n-1\} \big\}$.

Consider the following algorithm A': For each $\alpha \in F$, we create an instance with integral arrival times

$$a_\alpha(x_i) = \begin{cases} \lceil a(x_i) \rceil & \text{if } \operatorname{fp}(a(x_i)) > \alpha \\ \lfloor a(x_i) \rfloor & \text{otherwise} \end{cases} \qquad \text{for each } i \in \{0, \ldots, m-1\}. \tag{5.1}$$

For every $\alpha \in F$, we apply algorithm A to the instance with modified arrival times $a_\alpha(x_i) \in \mathbb{N}$ to obtain a circuit $C_\alpha \in S$ realizing f. We output a circuit C_{α^*} which has best delay with respect to arrival times a among all circuits created.

As $|F| \leq n$ and the running time of algorithm A is $r(n)$ by assumption, the running time algorithm A' is at most $\mathcal{O}(r(n)n)$.

In order to prove S-optimality of A', for each $\alpha \in F$, also consider the instance with arrival times $\tilde{a}_\alpha(x_i)$ resulting from $a(x_i)$ by rounding up to the next fractional number b_i with $\operatorname{fp}(b_i) = \alpha$. Note that for each $i \in \{0, \ldots, n-1\}$, we have

$$a_\alpha(x_i) \quad = \quad \tilde{a}_\alpha(x_i) - \alpha\,. \tag{5.2}$$

This implies

$$\operatorname{delay}(C_\alpha; a) \overset{a \leq \tilde{a}_\alpha}{\leq} \operatorname{delay}(C_\alpha; \tilde{a}_\alpha) \overset{(5.2)}{=} \operatorname{delay}(C_\alpha; a_\alpha) + \alpha\,. \tag{5.3}$$

Now, consider a circuit $C \in S$ with optimum delay $d := d(C; a) \in \mathbb{R}$ among all circuits in S. Consider an input x_j for which there is a path $P\colon x_j \rightsquigarrow \operatorname{out}(C)$ with $a(x_j) + |P| = d$. In particular, we have $\operatorname{fp}(x_j) = \operatorname{fp}(d)$. Let $\alpha^* := \operatorname{fp}(x_j)$. We will show that C_{α^*} fulfills $\operatorname{delay}(C_{\alpha^*}; a) = d$.

Note that $\operatorname{delay}(C; \tilde{a}_{\alpha^*}) = d$. Furthermore, we have

$$\operatorname{delay}(C; a_{\alpha^*}) \overset{(5.2)}{=} \operatorname{delay}(C; \tilde{a}_{\alpha^*}) - \alpha^* = d - \alpha^*\,. \tag{5.4}$$

As the circuit C is S-optimum for arrival times a, and, by S-optimality of algorithm A, the circuit C_{α^*} is S-optimum for arrival times a_{α^*}, we obtain

$$d \leq \operatorname{delay}(C_{\alpha^*}; a) \overset{(5.3)}{\leq} \operatorname{delay}(C_{\alpha^*}; a_{\alpha^*}) + \alpha^* \leq \operatorname{delay}(C; a_{\alpha^*}) + \alpha^* \overset{(5.4)}{=} d\,. \tag{5.5}$$

So in fact, all inequalities in Equation (5.5) are fulfilled with equality and C_{α^*} is S-optimum with respect to arrival times a. Hence, our algorithm A' is S-optimum. $\square$

For improving the running time of algorithm A', we need the essential property that algorithm A is monotone in the following sense:

Observation 5.1.4. Consider again the proof of Proposition 5.1.3, in particular the set $F := \{\operatorname{fp}(a(x_i)) : i \in \{0, \ldots, m\}\}$ and the modified arrival times a_α defined in Equation (5.1) for each $\alpha \in F$. Then, for $\alpha_1, \alpha_2 \in F$ with $\alpha_1 < \alpha_2$, we have $a_{\alpha_1}(x_i) \geq a_{\alpha_2}(x_i) \geq a_{\alpha_1}(x_i) - 1$ for each $i \in \{0, \ldots, m-1\}$. As in the proof, let $C_{\alpha_1}, C_{\alpha_2} \in S$ be the S-optimum circuits for arrival times a_{α_1} and a_{α_2} computed by algorithm A, respectively. By Lemma 5.1.2, (i), the circuit C_{α_1} is S-optimum for arrival times $a_{\alpha_1}(x_i) - 1$. Note that $\operatorname{delay}(C_{\alpha_1}; a_{\alpha_1} - 1) = \operatorname{delay}(C_{\alpha_1}; a_{\alpha_1}) - 1$. By Lemma 5.1.2, (ii), we thus have

$$\operatorname{delay}(C_{\alpha_1}; a_{\alpha_1}) \geq \operatorname{delay}(C_{\alpha_2}; a_{\alpha_2}) \geq \operatorname{delay}(C_{\alpha_1}; a_{\alpha_1}) - 1\,.$$

Using this observation, we can derive our final algorithm for instances with fractional arrival times.

α	0.0	0.3	0.4	0.5	0.7	0.8
$\text{delay}(C_\alpha; a_\alpha)$	6	6	6	5	5	5
$\text{delay}(C_\alpha; a_\alpha) + \alpha$	6.0	6.3	6.4	5.5	5.7	5.8

Table 5.1: Example illustrating the proof of Theorem 5.1.5. We have $\alpha^* = 0.5$.

Theorem 5.1.5. *Consider a Boolean function $f\colon \{0,1\}^n \to \{0,1\}$ with input variables $x_0, \ldots, x_{n-1}$ and a non-empty set S of circuits for f. Assume that algorithm A is an S-optimum algorithm with running time $\mathcal{O}(r(n))$ which is only defined for integral input arrival times. Then, there is an S-optimum algorithm A' for arbitrary fractional input arrival times with running time $\mathcal{O}(r(n)\log_2 n)$. We call A' the* ***binary-search extension*** *of algorithm A.*

Proof. We use the same notation as in the proof of Proposition 5.1.3. Denote the elements of F by $\alpha_0, \ldots, \alpha_{|F|-1}$, where $\alpha_j < \alpha_{j+1}$ for every $j \in \{0, \ldots, |F|-2\}$. Define the function

$$d_{\text{int}}\colon F \to \mathbb{N},\ \alpha \mapsto \text{delay}(C_\alpha; a_\alpha)\,.$$

By Observation 5.1.4, the function d_{int} is monotonely decreasing in α and assumes at most two values.

By Proposition 5.1.3, there is $\alpha^* \in F$ such that C_{α^*} is S-optimum for arrival times a. By Equation (5.3), for all $\alpha \in F$, we have $\text{delay}(C_\alpha; a) \leq \text{delay}(C_\alpha; a_\alpha) + \alpha$, and by Equation (5.5), equality holds for $\alpha = \alpha^*$. Hence, we have

$$\alpha^* = \min\left\{ \alpha \in F : d_{\text{int}}(\alpha) = d_{\text{int}}\Big(\alpha_{|F|-1}\Big) \right\}.$$

Using binary search on F, we can determine α^* in $\mathcal{O}(\log_2 |F|) = \mathcal{O}(\log_2 m)$ steps. Table 5.1 shows an example how to find α^*. □

5.2 Structure Theorem

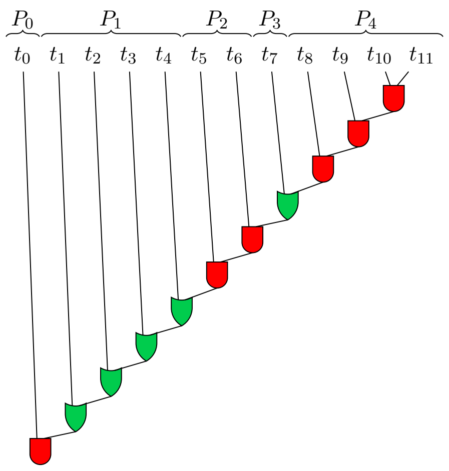
Our structure theorem and our algorithm presented in Section 5.3 both reduce the problem of optimizing a given generalized AND-OR path to smaller instances of a specific form. For introducing these generalized AND-OR paths, recall from Definition 2.5.6 that a generalized AND-OR path comes with a signal partition, i.e., a unique partition of the inputs into maximal disjoint sets that are all propagate or all generate signals.

Notation 5.2.1. Let $h(t;\Gamma)$ with $t = (t_0, \ldots, t_{m-1})$ and $\Gamma = (\circ_0, \ldots, \circ_{m-2})$ be a generalized AND-OR path with signal partition $(t_0, \ldots, t_{m-1}) = P_0 +\!\!+ \ldots +\!\!+ P_c$. Let $b, b' \in \{0, \ldots, c\}$ with $b \leq b'$.

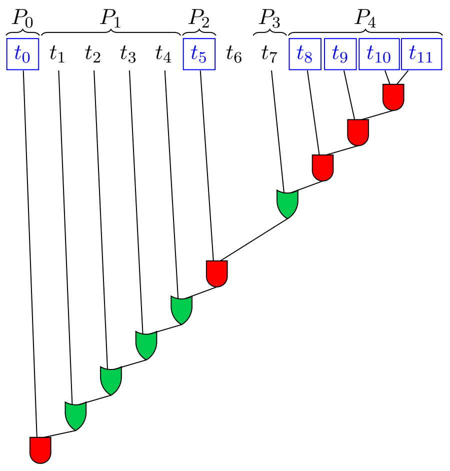
We denote the minimum index $i \in \{0, \ldots, m-1\}$ with $t_i \in P_b$ by $l(b)$ and the maximum index $i \in \{0, \ldots, m-1\}$ with $t_i \in P_b$ by $r(b)$.

For an n-tuple $(x_0, \ldots, x_{n-1})$ and an index $i \in \{0, \ldots, n-1\}$, we use the standard notation $(x_0, \ldots, \widehat{x_i}, \ldots, x_{n-1})$ to denote the $(n-1)$-tuple arising from x by deleting the entry x_i. Given $i \in \{0, \ldots, m-1\}$, we define $h(t;\Gamma)_{\widehat{t_i}}$ as the generalized AND-OR path arising from $h(t;\Gamma)$ by removing t_i, i.e.,

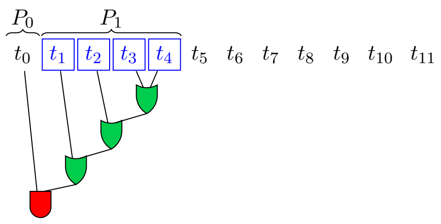
$$h(t;\Gamma)_{\widehat{t_i}} := \begin{cases} h\Big((t_0, \ldots, \widehat{t_i}, \ldots, t_{m-1}); (\circ_0, \ldots, \widehat{\circ_i}, \ldots, \circ_{m-2})\Big) & \text{if } i \leq m-2\,, \\ h\Big((t_0, \ldots, t_{m-2}); (\circ_0, \ldots, \circ_{m-3})\Big) & \text{if } i = m-1\,. \end{cases}$$


P_0
P_1
P_2
P_3
P_4
t_0 t_1 t_2 t_3 t_4 t_5 t_6 t_7 t_8 t_9 t_{10} t_{11}

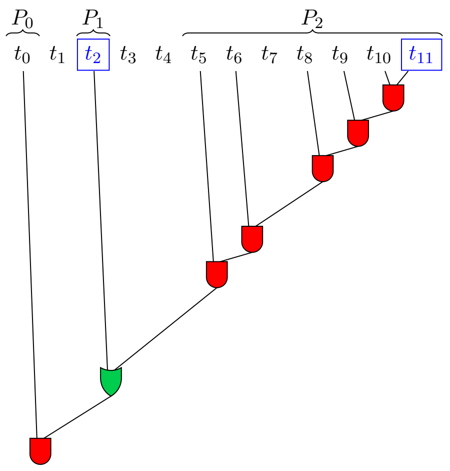
(a) Standard circuit for $h(t;\Gamma)$ with $S^{\textsc{And}} = \{t_0, t_5, t_6, t_8, t_9, t_{10}, t_{11}\}$ and $S^{\textsc{Or}} = \{t_1, t_2, t_3, t_4, t_7, t_{11}\}$.


P_0
P_1
P_2
P_3
P_4
t_0 t_1 t_2 t_3 t_4 t_5 t_6 t_7 t_8 t_9 t_{10} t_{11}

(b) Standard circuit for $h(t;\Gamma)_{S_1^{\textsc{And}}} = h(t;\Gamma)_{\widehat{t_6}}$ with $S_1^{\textsc{And}} = \{t_0, t_5, t_8, t_9, t_{10}, t_{11}\}$ and $S_2^{\textsc{And}} = \{t_6\}$.


P_0
P_1
t_0 t_1 t_2 t_3 t_4 t_5 t_6 t_7 t_8 t_9 t_{10} t_{11}

(c) Std. circuit for $h(t;\Gamma)_{S_1^{\textsc{Or}}} = h(t;\Gamma)_{[0:4]}$ with $S_1^{\textsc{Or}} = \{t_1, t_2, t_3, t_4\}$, $S_2^{\textsc{Or}} = \{t_7, t_{11}\}$.


P_0
P_1
P_2
t_0 t_1 t_2 t_3 t_4 t_5 t_6 t_7 t_8 t_9 t_{10} t_{11}

(d) Standard circuit for $h(t;\Gamma)_{S_1^{\textsc{Or}}}$ with $S_1^{\textsc{Or}} = \{t_2, t_{11}\}$ and $S_2^{\textsc{Or}} = \{t_1, t_3, t_4, t_7\}$.

Figure 5.1: Circuits for several generalized And-Or paths on subsets of $t = (t_0, \dots, t_{11})$ arising from the generalized And-Or path $h(t;\Gamma)$ from Figure 5.1(a) as in Notation 5.2.1. We also show the respective signal partitions, and the respective input set S_1° is marked blue.

We extend this notation to removal of a subset $F = \left\{ t_{i_0}, \dots, t_{i_{f-1}} \right\}$ of inputs by

$$h(t;\Gamma)_{\widehat{F}} := \left(\left(h(t;\Gamma)_{\widehat{t_{i_0}}} \right)_{\widehat{t_{i_1}}} \right)_{\dots}$$

Furthermore, given $0 \leq i \leq j \leq m-1$, we define $h(t;\Gamma)_{[i:j]}$ as the generalized And-Or path arising from $h(t;\Gamma)$ by deleting all inputs t_k with $k < i$ or $k > j$.

Given a gate type $\circ \in \{\textsc{And}, \textsc{Or}\}$, we denote the set of all signals t_i with $\circ_i = \circ$ plus t_{m-1} by S°. We call this set the **same-gate input set** of the generalized And-

Or path $h(t;\Gamma)$ and the gate type $\circ$. Consider a partition $S^\circ = S_1^\circ \uplus S_2^\circ$ with $S_1^\circ \neq \emptyset$. Let $i_1 \in \{0, \dots, m-1\}$ be maximum with $t_{i_1} \in S_1^\circ$. We write

$$h(t;\Gamma)_{S_1^\circ} := \Big(h(t;\Gamma)_{[0:i_1]}\Big)_{\widehat{S_2^\circ}}.$$

Note that the same-gate input set S° always contains the input t_{m-1}. In the entire chapter, t_{m-1} will play a special role among the inputs.

Figure 5.1 shows a generalized And-Or path and gives several examples of smaller generalized And-Or paths arising from a partition of S^{And} or S^{Or}.

Observation 5.2.2. Let $h(t;\Gamma)$ with $t = (t_0, \dots, t_{m-1})$ and $\Gamma = (\circ_0, \dots, \circ_{m-2})$ be given. Consider a gate type $\circ \in \{\text{And}, \text{Or}\}$, and a partition $S^\circ = S_1^\circ \uplus S_2^\circ$ of the same-gate input set of $h(t;\Gamma)$ with $S_1^\circ, S_2^\circ \neq \emptyset$. Let $i_k \in \{0, \dots, m-1\}$ be maximum with $t_{i_k} \in S_k^\circ$. Then, $h(t;\Gamma)_{S_k^\circ}$ is a generalized And-Or path depending essentially on exactly the inputs from S_k° plus all signals t_j with $\circ_j \neq \circ$ and $j < i_k$. Furthermore, for every $i < i_k$ such that $h(t;\Gamma)_{S_k^\circ}$ depends essentially on t_i, the signal t_i is a propagate signal (generate signal) of $h(t;\Gamma)_{S_k^\circ}$ if and only if it is a propagate signal (generate signal) of $h(t;\Gamma)$.

In Theorem 5.2.9, we will see that for any generalized And-Or path $h(t;\Gamma)$ with input arrival times, there is always a delay-optimum formula circuit C which arises from a partition $S^\circ = S_1^\circ \uplus S_2^\circ$ into non-empty subsets.

But first, in Proposition 5.2.4, we give basic insights on the structure of any circuit realizing a given generalized And-Or path. For this, we need the following observation. It can be seen easily by considering the standard realization for $h(t;\Gamma)$ as C (which is possible because a Boolean function is independent of the realizing circuit). Figures 5.1(a) to 5.1(c) illustrate this observation in the case that $i = 6$ and t_i is a propagate signal.

Observation 5.2.3. Let $h(t;\Gamma)$ with $t = (t_0, \dots, t_{m-1})$ and $\Gamma = (\circ_0, \dots, \circ_{m-2})$ be a generalized And-Or path. Let C be a circuit realizing $h(t;\Gamma)$. Assume that t_i is an input with $t_i \in P_b$ for some $b \in \{0, \dots, c\}$, and consider $\alpha \in \{0, 1\}$.

(i) If t_i is a generate signal and $\alpha = 0$, or if t_i is a propagate signal and $\alpha = 1$, then we have $f\big(C|_{t_i=\alpha}\big) = h(t;\Gamma)_{\widehat{t_i}}$. In particular, $f(C|_{t_i=\alpha})$ depends essentially on $t_0, \dots, t_{i-1}, t_{i+1}, \dots, t_{m-1}$.

(ii) Assume that $b > 0$. If t_i is a propagate signal and $\alpha = 0$, or if t_i is a generate signal and $\alpha = 1$, then we have $f(C|_{t_i=\alpha}) = h(t;\Gamma)_{[0:r(b-1)]}$. In particular, $f(C|_{t_i=\alpha})$ depends essentially on $t_0, \dots, t_{r(b-1)}$ and does not depend essentially on $t_{l(b)}, \dots, t_{m-1}$.

Proposition 5.2.4. *Let $h(t;\Gamma)$ with $t = (t_0, \dots, t_{m-1})$ and $\Gamma = (\circ_0, \dots, \circ_{m-2})$ be a generalized* And-Or *path with signal partition $(t_0, \dots, t_{m-1}) = P_0 \mathbin{+\!\!+} \dots \mathbin{+\!\!+} P_c$. Consider a circuit C realizing $h(t;\Gamma)$. Then, the following statements hold:*

(i) Any input t_i is connected to the output.

(ii) If $m \geq 2$, any input t_i has depth at least 1 in C.

(iii) For input $t_i \in P_b$ with $b > 0$, each directed path Q from t_i to $\text{out}(C)$ *contains at least one* And *gate and at least one* Or *gate. In particular, t_i has depth at least 2 in C.*

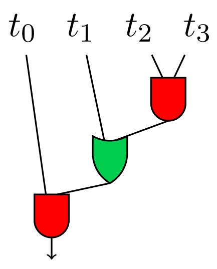


Figure 5.2: A depth-optimum AND-OR path on 4 inputs.

Proof. From the fact that generalized AND-OR paths depend essentially on all their inputs (cf. Corollary 2.5.9), the first two statements follow immediately. To see the third statement, note that by Observation 5.2.3, for any $t_i \in P_b$ with $b > 0$ and for any $\alpha \in \{0, 1\}$, the function $f(C\,|_{t_i=\alpha})$ depends essentially on t_0. This is not the case if some directed path from t_i to $\text{out}(C)$ contains only gates of the same type. $\square$

From this proposition, we can derive the following lower bound on the delay of any circuit for a given generalize AND-OR path that we will use in our algorithm in Section 5.3.

Corollary 5.2.5. *Let $m \in \mathbb{N}$ with $m \geq 2$. Let inputs $t = (t_0, \dots, t_{m-1})$ with arrival times $a(t_0), \dots, a(t_{m-1}) \in \mathbb{R}$ and gate types $\Gamma = (\circ_0, \dots, \circ_{m-2})$ be given. Let $(t_0, \dots, t_{m-1}) = P_0 \mathbin{+\!\!+} \dots \mathbin{+\!\!+} P_c$ be the signal partition of $h(t; \Gamma)$. Consider a circuit C realizing $h(t; \Gamma)$. Then, we have*

$$\text{delay}(C) \geq \max\left\{ \max_{t_i \in P_0} a(t_i) + 1, \max_{t_i \in P_b : b > 0} a(t_i) + 2 \right\}.$$

A simple application of this statement is shown now.

Proposition 5.2.6. *For $1 \leq m \leq 3$, the standard* AND-OR *path circuit for $g\big((t_0, \dots, t_{m-1})\big)$ is delay-optimum regardless of the input arrival times. The standard* AND-OR *path circuit for $g\big((t_0, \dots, t_3)\big)$ is depth-optimum.*

Proof. For $m = 1$, the statement is trivial.

For $m \in \{2, 3\}$, the statement follows directly from Corollary 5.2.5.

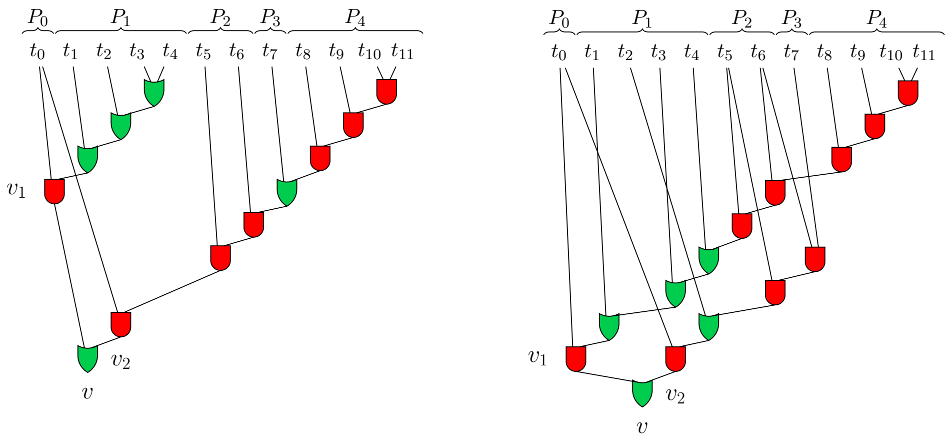
Now let $m = 4$ and $g(t) = t_0 \wedge \big(t_1 \vee (t_2 \wedge t_3)\big)$. Figure 5.2 depicts the standard circuit for $g(t)$. We claim that its depth of 3 is optimum for a circuit realizing $g(t)$, so assume there is a circuit C for $g(t)$ with depth 2.

By Proposition 5.2.4, (iii), any directed path from t_2 or t_3 to $\text{out}(C)$ contains exactly one AND gate and exactly one OR gate. Let $\alpha \in \{0, 1\}$ be given by 0 if $\text{out}(C) = $ AND and by 1 otherwise, and let $\beta \in \{0, 1\}$ be given by $\beta = \overline{\alpha}$.

Now, if t_2 and t_3 have a common direct successor vertex $v \in \mathcal{G}(C)$, we have $C\,|_{t_2=t_3=\alpha} = \alpha$. But in Figure 5.2, we can see that $g(t)\,|_{t_2=t_3=\alpha}$ depends essentially on t_1 if $\alpha = 0$ and on t_0 if $\alpha = 1$, so this is a contradiction.

Thus, t_2 and t_3 both have fanout 1 and have different successor vertices $v \neq w \in \mathcal{G}(C)$ with $v, w \neq \text{out}(C)$ and $\text{gt}(v) = \text{gt}(w)$. As C depends essentially on t_0 and t_1, the second inputs of v and w are t_0 and t_1, and as $\text{gt}(v) = \text{gt}(w)$, we may assume that $t_0 \in \delta^-(v)$ and $t_1 \in \delta^-(w)$. But then, we have $C\,|_{t_2=t_3=\beta} = \beta$, again a contradiction to the fact that $C\,|_{t_2=t_3=\beta}$ depends essentially on t_1 or t_0.

Hence, the standard AND-OR path circuit is depth-optimum for $m = 4$. $\square$


P_0 P_1 P_2 P_3 P_4
t_0 t_1 t_2 t_3 t_4 t_5 t_6 t_7 t_8 t_9 t_{10} t_{11}
v_1
v_2
v
P_0 P_1 P_2 P_3 P_4
t_0 t_1 t_2 t_3 t_4 t_5 t_6 t_7 t_8 t_9 t_{10} t_{11}
v_1
v_2
v

(a) Here, we have $f(C_{v_1}) = h(t;\Gamma)_{S_k^\circ}$ with $S_1^{\text{OR}} = \{t_1, \dots, t_4\}$ and $f(C_{v_2}) = h(t;\Gamma)_{S_2^\circ}$ with $S_2^{\text{OR}} = \{t_7, t_{11}\}$.

(b) Here, we have $f(C_{v_1}) = h(t;\Gamma)_{S_k^\circ}$ with $S_1^{\text{OR}} = \{t_1, t_3, t_4, t_{11}\}$ and $f(C_{v_2}) = h(t;\Gamma)_{S_2^\circ}$ with $S_2^{\text{OR}} = \{t_2, t_7\}$.

Figure 5.3: Two possible circuits arising from Lemma 5.2.8 applied to the generalized AND-OR path from Figure 5.1(a).

From now on, we restrict ourselves to formula circuits. Our idea for analyzing the structure of delay-optimum formula circuits realizing a given generalized AND-OR path is based on Lemma 1 from Commentz-Walter [Com79]. However, she only considers AND-OR paths, and not generalized AND-OR paths, and gives only a partial description of the structure of AND-OR path circuits, not a complete characterization. The main objects considered in both her and our proof are prime implicants.

Observation 5.2.7. Let a generalized AND-OR path $h(t;\Gamma)$ on input variables $t_0, \dots, t_{m-1}$ and gate types $\Gamma = (\circ_0, \dots, \circ_{m-2})$ be given. Recall from Notation 5.2.1 that S^{OR} contains all input variables t_i that are generate signals or fulfill $i = m-1$. Hence, by Corollary 2.5.8, the prime implicants of $h(t;\Gamma)$ are given by

$$\left\{ t_i \wedge \bigwedge_{j<i, t_j \text{ propagate signal}} t_j : t_i \in S^{\text{OR}} \right\}.$$

Now consider any two prime implicants $\pi, \rho \in \text{PI}(h(t;\Gamma))$ with $\pi \neq \rho$. Let $i \in \{0, \dots, m-1\}$ maximum with $t_i \in \text{lit}(\pi)$. Then, we have $t_i \notin \text{lit}(\rho)$.

The following lemma is the main ingredient of our structure theorem, Theorem 5.2.9. We consider a formula circuit C implementing a generalized AND-OR path h with $\text{out}(C) = \text{OR}$. If C is delay-optimum for given arrival times and, among all delay-optimum circuits, size-optimum, then we will show that the two sub-circuits of $\text{out}(C)$ are again generalized AND-OR paths on a subset of the inputs, where each input of S^{OR} is contained in exactly one of the two sub-circuits. Figure 5.3 shows two examples for such circuits for the generalized AND-OR path from Figure 5.1(a).

Lemma 5.2.8. *Let $m \in \mathbb{N}_{\geq 2}$, inputs $t = (t_0, \dots, t_{m-1})$ with arrival times $a(t_0), \dots, a(t_{m-1}) \in \mathbb{R}$ and gate types $\Gamma = (\circ_0, \dots, \circ_{m-2})$ be given. Consider a delay-optimum formula circuit C for $h(t;\Gamma)$ with minimum number of gates. Assume that*

$\text{gt}(\text{out}(C)) = \textsc{Or}$*, and let* $S^{\textsc{Or}}$ *be the same-gate input set of* h *and* $\textsc{Or}$*. Denote the predecessors of* $v := \text{out}(C)$ *by* v_1 *and* v_2*. Write* $h := h(t;\Gamma)$*, and* $f_1 := f(C_{v_1})$*, and* $f_2 := f(C_{v_2})$*. Then, the following statements are fulfilled:*

(i) *We have* $\text{PI}(h) = \text{PI}(f_1) \mathbin{\dot\cup} \text{PI}(f_2)$*.*

(ii) *There is a partition* $S^{\textsc{Or}} = S_1^{\textsc{Or}} \mathbin{\dot\cup} S_2^{\textsc{Or}}$ *with* $S_1^{\textsc{Or}}, S_2^{\textsc{Or}} \neq \emptyset$ *such that for all* $k, l \in \{1,2\}$ *with* $k \neq l$*, the function* f_k *depends essentially on all inputs of* $S_k^{\textsc{Or}}$ *and on no input of* $S_l^{\textsc{Or}}$*.*

(iii) *Let* $k \in \{1,2\}$*. Consider the generalized* $\textsc{And}$*-*$\textsc{Or}$ *path* $h_k := h(t;\Gamma)_{S_k^{\textsc{Or}}}$*. Then, we have* $f_k = h_k$*.*

Proof. As $\text{gt}(\text{out}(C)) = \textsc{Or}$, by Lemma 2.1.27,

(a) any implicant of f_1 or f_2 is an implicant of h, and

(b) any prime implicant of h is a prime implicant of f_1 or f_2.

Item (b) implies $\text{PI}(h) \subseteq \text{PI}(f_1) \cup \text{PI}(f_2)$. It remains to prove that $\text{PI}(f_k) \subseteq \text{PI}(h)$ for each $k \in \{1,2\}$ and that $\text{PI}(f_1) \cap \text{PI}(f_2) = \emptyset$.

By Item (a), any prime implicant ρ of f_1 is an implicant of h and must hence contain a prime implicant π of h. By Item (b) and the definition of prime implicants, we have $\rho = \pi$, or π is a prime implicant of f_2. Note that $\rho = \pi$ would imply $\rho \in \text{PI}(h)$). Hence, to prove the first statement, it suffices to show the following claim.

Claim. If there are $\rho \in \text{PI}(f_1)$ and $\pi \in \text{PI}(h)$ with $\text{lit}(\pi) \subseteq \text{lit}(\rho)$ and $\pi \in \text{PI}(f_2)$, then C is not a size-minimum delay-optimum circuit for h.

Proof of claim: Choose $i \in \{0, \dots, m-1\}$ maximum such that t_i is contained in π.

As C is a formula circuit, we have $\mathcal{G}(C_1) \cap \mathcal{G}(C_2) = \emptyset$. Consider the circuit B arising from C by replacing C_1 with the reduced circuit $C_1 \mid_{t_i=0}$. Note that B is again a formula circuit. Write $g := f(B)$, and for $k \in \{1,2\}$, write $B_k := B_{v_k}$ and $g_k := f(B_k)$. As ρ contains t_i, by Observation 2.1.20, f_1 depends essentially on t_i. Hence, by Observation 2.3.14, we have $\text{delay}(B) \le \text{delay}(C)$ and $\text{size}(B) < \text{size}(C)$. It remains to show that B and C are logically equivalent.

Let $\alpha \in \{0,1\}^m$. As B is monotone and arises from C by fixing an input to 0, we have $g(\alpha) = 0$ whenever $h(\alpha) = 0$. Thus, assume that $h(\alpha) = 1$. Then, there is $\psi \in \text{PI}(h)$ with $\psi(\alpha) = 1$.

Case 1: We have $\psi \in \text{PI}(f_2)$.

Here, as $B_2 = C_2$, we have $g_2(\alpha) = f_2(\alpha) = 1$ and thus $g(\alpha) = 1$.

Case 2: We have $\psi \notin \text{PI}(f_2)$.

By Item (b), we have $\psi \in \text{PI}(f_1)$. As $\pi \in \text{PI}(f_2)$, we must have $\psi \neq \pi$. By the choice of t_i, Observation 5.2.7 implies that $t_i \notin \text{lit}(\psi)$. As any implicant ι of f_1 with $t_i \notin \text{lit}(\iota)$ is an implicant of g_1, we have $\psi \in \text{PI}(g_1)$. This implies $g(\alpha) = 1$.

Thus, B is a delay-optimum formula circuit for h with better size than C. □

Now, we show the second statement. For each $k \in \{1,2\}$, let $S_k^{\textsc{Or}}$ consist of the inputs among $S^{\textsc{Or}}$ that f_k depends on essentially. By Observation 2.1.20, a Boolean function depends essentially on an input t_i if and only if t_i is contained in any of its prime implicants. By Observation 5.2.7, for each input $t_i \in S^{\textsc{Or}}$, there is exactly one prime implicant of h containing t_i. Thus, we have $S^{\textsc{Or}} = S_1^{\textsc{Or}} \mathbin{\dot\cup} S_2^{\textsc{Or}}$.

Now, assume the conditions of the third statement. Additionally, let $i_k \in \{0, \dots, m-1\}$ maximum with $t_{i_k} \in S_k^{\text{OR}}$. By Corollary 2.5.8 and the first two statements, the prime implicants of f_k are

$$\left\{ t_i \wedge \bigwedge_{j<i, t_j \text{ propagate signal of } h} t_j : t_i \in S_k^{\text{OR}} \right\};$$

and by Corollary 2.5.8 and the definition of h_k, the prime implicants of h_k are

$$\left\{ t_i \wedge \bigwedge_{j<i, t_j \text{ propagate signal of } h_k} t_j : t_i \text{ generate signal of } h_k \text{ or } i = i_k \right\}.$$

Note that both for f_k and for h_k, the maximum index of any essential input is i_k. By Observation 5.2.2, an input t_i of h_k with $i < i_k$ is a generate signal (propagate signal) of h_k if and only if it is a generate signal (propagate signal) of h contained in S_k^{OR}. Moreover, $t_{i_k} \in S_k^{\text{OR}}$ by definition of i_k. Hence, we have $\text{PI}(f_k) = \text{PI}(h_k)$.

By Corollary 2.1.22, we deduce $h_k = f_k$, hence the third statement. □

Theorem 5.2.9 (Structure theorem). *Let inputs $t = (t_0, \dots, t_{m-1})$ with arrival times $a(t_0), \dots, a(t_{m-1}) \in \mathbb{R}$ and gate types $\Gamma = (\circ_0, \dots, \circ_{m-2})$ be given. Consider a delay-optimum formula circuit C for $h(t;\Gamma)$ with minimum number of gates. Let $\circ := \text{gt}(\text{out}(C))$ and let S° denote the same-gate input set for $h(t;\Gamma)$ and $\circ$. Denote the predecessors of $v := \text{out}(C)$ by v_1 and v_2. Write $f_1 := f(C_{v_1})$, and $f_2 := f(C_{v_2})$. Then, there is a partition $S^\circ = S_1^\circ \mathbin{\dot\cup} S_2^\circ$ into non-empty subsets such that for each $k, l \in \{1, 2\}$ with $l \neq k$, the function f_k depends essentially on the inputs of S_k°, but not on those of S_l°. Moreover, for each $k \in \{1, 2\}$, we have*

$$f_k = h(t;\Gamma)_{S_k^\circ}.$$

Proof. By duality, it suffices to consider the case $\text{gt}(\text{out}(C)) = \text{OR}$. In this case, the statements hold by Lemma 5.2.8. □

As a consequence of this theorem, we can derive an upper bound on the maximum number of inputs an AND-OR path may have such that an AND-OR path circuit with fixed depth d exists. Recall that this number is denoted by $m(d, 0)$ in Definition 3.1.1. The upper bound presented in the following corollary can be seen easily; probably, much stronger bounds can be derived from Theorem 5.2.9.

Corollary 5.2.10. *Let $d \in \mathbb{N}_{\geq 1}$ be given. We have $m(d+1, 0) \leq 2m(d, 0)$.*

Proof. Let $m \in \mathbb{N}$ maximum such that an AND-OR path $h(t)$ on m inputs $t = (t_0, \dots, t_{m-1})$ can be realized by a circuit with depth $d+1$. As $d \geq 1$, we have $m \geq 3$. Consider a formula circuit C for $h(t)$ with depth $d+1$ and minimum number of gates. We need to show that $m \leq 2m(d, 0)$.

Dualization allows us to assume that $\text{out}(C) = \text{OR}$. We apply the structure theorem, Theorem 5.2.9, to C. This implies that there are circuits C_1 and C_2 with depth at most d that realize generalized AND-OR paths f_1 and f_2, respectively, such that $C = C_1 \vee C_2$. Consider the partition $S^{\text{OR}} = S_1^{\text{OR}} \mathbin{\dot\cup} S_2^{\text{OR}}$ of the same-gate signals of $h(t)$ as in Theorem 5.2.9.

Let $D^{\textsc{Or}} := \{t_0, \dots, t_{m-1}\} \backslash S^{\textsc{Or}}$. As $h(t)$ is an And-Or path and $m \geq 3$, we have $D^{\textsc{Or}} \neq \emptyset$. As $h(t)$ is an And-Or path and $t_{m-1} \in S^{\textsc{Or}}$, for every $t_i \in D^{\textsc{Or}}$, we have $t_{i+1} \in S^{\textsc{Or}}$. Hence, the function

$$\vartheta \colon D^{\textsc{Or}} \to S^{\textsc{Or}}, \quad t_i \mapsto t_{i+1}$$

is well-defined. For $k \in \{1, 2\}$, let $D_k^{\textsc{Or}} := \vartheta^{-1}\Big(S_k^{\textsc{Or}}\Big)$. Note that $D^{\textsc{Or}} = D_1^{\textsc{Or}} \dot\cup D_2^{\textsc{Or}}$.

Now, for each $k \neq l \in \{1, 2\}$, let B_k denote the reduced circuit arising from C_k by fixing all inputs $t_i \in D_l$ to $\alpha := 1$, and let $g_k := f(B_k)$. Then, as all inputs in D_l are propagate signals, by considering the standard circuit for $h(t)$, we observe that $g_k = (f_k)_{\widehat{D_l^{\textsc{Or}}}}$. By construction, the essential variables of g_k are the variables of $S_k^{\textsc{Or}}$ and $D_k^{\textsc{Or}}$. Let t_{j_k} be the essential variable of g_k with j_k maximum.

Consider $k \in \{1, 2\}$. We show that g_k is an And-Or path: First note that by Observation 5.2.2 and the choice of α, every input of g_k except for t_{j_k} is a propagate signal (generate signal) of g_k if and only if it is a propagate signal (generate signal of $h(t)$. By definition of ϑ, for any two generate signals t_i, t_j of g_k with $i < j < j_k$, the propagate signal $t_{j-1} = \vartheta^{-1}(t_j)$ of $h(t)$ is an input of g_k. Furthermore, for any two propagate signals $t_i \neq t_j$ of g_k with $i < j < j_k$, the generate signal $t_{i+1} = \vartheta(t_i)$ of $h(t)$ is an input of g_k. Hence, the inputs of g_k (except for t_{j_k}) are alternatingly propagate and generate signals and g_k is an And-Or path.

Let m_1 and m_2 be the numbers of inputs of B_1 and B_2, respectively. As

$$\{t_0, \dots, t_{m-1}\} = S^{\textsc{Or}} \dot\cup D^{\textsc{Or}} = S_1^{\textsc{Or}} \dot\cup S_2^{\textsc{Or}} \dot\cup D_1^{\textsc{Or}} \dot\cup D_2^{\textsc{Or}} ,$$

we have $m_1 + m_2 = m$. As B_1 and B_2 are both And-Or path circuits with depth at most d, we have $m_1, m_2 \leq m(d, 0)$. Together, this implies

$$m = m_1 + m_2 \leq 2m(d, 0) . \qquad \square$$

For the special case when all input arrival times are equal, we conjecture that partitions of the same-gate inputs into two "non-overlapping" sets are always best for the delay.

Conjecture 5.2.11. *Consider Theorem 5.2.9 for the case of uniform input arrival times and let $S^\circ = S_1^\circ \dot\cup S_2^\circ$ be a partition as in the theorem. Then, for all inputs $t_i \in S_1^\circ$ and $t_j \in S_2^\circ$, we have $i < j$.*

We will see in Section 6.3 why we assume this statement to be satisfied. For non-uniform arrival times, we already know that the conjecture is not fulfilled, see Figure 6.12 (page 189).

5.3 General Algorithm

The structure theorem from the previous section motivates an exact algorithm for the Generalized And-Or Path Circuit Optimization Problem: Consider a generalized And-Or path $h(t; \Gamma)$ with prescribed input arrival times. Assume that we know a delay-optimum formula circuit for all generalized And-Or paths on strict sub-vectors of t. Then, by Theorem 5.2.9, for $\circ$ = And or for $\circ$ = Or, there is a partition $S^\circ = S_1^\circ \dot\cup S_2^\circ$ such that $C := h(t; \Gamma)_{S_1^\circ} \circ h(t; \Gamma)_{S_2^\circ}$ is a delay-optimum formula circuit for $h(t; \Gamma)$.

There is a map κ from the sets of generalized And-Or paths arising from recursive applications of the structure theorem to the non-empty subsets of $t_0, \dots, t_{m-1}$

Algorithm 5.1: Exact algorithm for delay optimization of generalized AND-OR paths

```
Input: Boolean input variables $t = (t_0, \dots, t_{m-1})$ with arrival times
       $a(t_0), \dots, a(t_{m-1}) \in \mathbb{R}$, and gate types $\Gamma = (\circ_0, \dots, \circ_{m-2})$.
Output: Optimum delay of any circuit over $\Omega_{\text{mon}}$ computing $h(t; \Gamma)$.
1  foreach $\emptyset \neq I \subseteq \{t_0, \dots, t_{m-1}\}$ do
2      Set $d(I) := \infty$.
3  return compute_opt($\{t_0, \dots, t_{m-1}\}$)
   // Assume that $\emptyset \neq I \subseteq \{t_0, \dots, t_{m-1}\}$.
4  procedure compute_opt($I$)
5      Assume that $I = \{t_{i_0}, \dots, t_{i_{r-1}}\}$ with $0 \leq i_0 < \dots < i_{r-1} \leq m-1$ and
       let $\Gamma' := (\circ_{i_0}, \dots, \circ_{i_{r-2}})$.
6      if $d(I) < \infty$ then
7          return $d(I)$
8      if $r = 1$ then
9          Set $d(I) = a(t_{i_{r-1}})$.
10         return $d(I)$
11     foreach $\circ \in \{\text{AND}, \text{OR}\}$ do
12         Let $S^\circ \subseteq I$ consist of all signals $t_{i_j}$ with $\circ_{i_j} = \circ$ and $t_{i_{r-1}}$.
13         foreach partition $S^\circ = S_1^\circ \cup S_2^\circ$ with $S_1^\circ, S_2^\circ \neq \emptyset$ do
14             foreach $k \in \{1, 2\}$ do
15                 Let $I_k$ denote the input set of $h\Big(\big(t_{i_0}, \dots, t_{i_{r-1}}\big); \Gamma'\Big)_{S_k^\circ}$.
16                 Let $d_k :=$ compute_opt($I_k$).
17             Set $d(I) = \min\{d(I), \max\{d_1, d_2\} + 1\}$.
18     return $d(I)$
```

which maps each generalized AND-OR path to its essential inputs. This map is injective as for $h(t;\Gamma)$ by Observation 5.2.2, every input t_i with $i < i_k$ (with i_k as in Observation 5.2.2) that is essential for the generalized AND-OR path $h(t;\Gamma)_{S_k^\circ}$ is a propagate signal (generate signal) of $h(t;\Gamma)_{S_k^\circ}$ if and only if it is a propagate signal (generate signal) of $h(t;\Gamma)$.

Hence, we may identify a generalized AND-OR path considered during recursive applications of Theorem 5.2.9 with the set of its essential inputs. Algorithm 5.1 describes our algorithm which recursively applies Theorem 5.2.9 and stores the computed delays $d(I)$ for subsets I of $\{t_0, \dots, t_{m-1}\}$ in a dynamic programming table of size at most $2^m - 1$.

It is not hard to see that κ is actually a bijection: Given some subset $\emptyset \neq I \subsetneq \{t_0, \dots, t_{m-1}\}$, we need to find a series of partitions according to Theorem 5.2.9 such that the generalized AND-OR path with essential inputs I arises. Choose $i \in \{0, \dots, m-1\}$ maximum with $t_i \in I$. Assume that t_i is a generate signal (the other case follows by duality). First, use an OR gate and partition the same-gate signals S^{OR} of $h(t;\Gamma)$ and OR into those contained in I and the rest. Then, $h(t;\Gamma)_{S^{\text{OR}} \cap I}$ is a generalized AND-OR path with the generate signals contained in I, plus all propagate signals t_j of h with $j < i$. Afterwards, partition the propagate signals of $h(t;\Gamma)_{S^{\text{OR}} \cap I}$ into those contained in I and the rest. This yields the generalized And-Or path with essential input set I.

In the following theorem, we estimate the running time of Algorithm 5.1.

Theorem 5.3.1. *Let input variables $t = (t_0, \dots, t_{m-1})$ with arrival times $a(t_0), \dots, a(t_{m-1}) \in \mathbb{R}$ and gate types $\Gamma = (\circ_0, \dots, \circ_{m-2})$ be given. Then, Algorithm 5.1 computes the optimum delay of any circuit realizing the generalized* And-Or *path $h(t;\Gamma)$. The dynamic programming table needed to store the delay of all generalized* And-Or *paths considered during the computation has $2^m - 1$ entries. Denoting by g and p the number of generate signals and propagate signals among $t_0, \dots, t_{m-2}$, the algorithm can be implemented to run in time $\mathcal{O}(3^g 2^p + 2^g 3^p)$. In particular, if $h(t;\Gamma)$ is an* And-Or *path, then the running time is $\mathcal{O}\Big(\big(\sqrt{6}\big)^n\Big)$. By backtracking, we can obtain a delay-optimum formula circuit for $h(t;\Gamma)$.*

Proof. We have already argued that a generalized And-Or path arising from recursive application of Theorem 5.2.9 can be identified with the set of its essential inputs via a bijection κ. Hence, by induction on m and Theorem 5.2.9, we can see that Algorithm 5.1 computes the optimum delay of any formula circuit for $h(t;\Gamma)$. By Theorem 2.3.11, this is the optimum delay of any circuit for $h(t;\Gamma)$. The dynamic programming table has size exactly $2^m - 1$.

Let $T := \{t_0, \dots, t_{m-1}\}$. The running time of Algorithm 5.1 is dominated by enumerating all partitions of the respective set S° in line 13 for the two cases that $\circ =$ And or $\circ =$ Or for all subsets $\emptyset \neq I \subseteq T$. A partition of S° into 2 non-empty subsets corresponds to choosing a subset $S_1^\circ \subseteq S^\circ \backslash \{t_{i_k}\}$ and setting $S_2^\circ := S^\circ \backslash S_1^\circ$. By Observation 5.2.2, the generalized And-Or paths $h(t;\Gamma)_{S_1^{\text{Or}}}$ and $h(t;\Gamma)_{S_2^{\text{Or}}}$ are uniquely determined by I, S^{Or} and S_1^{Or}.

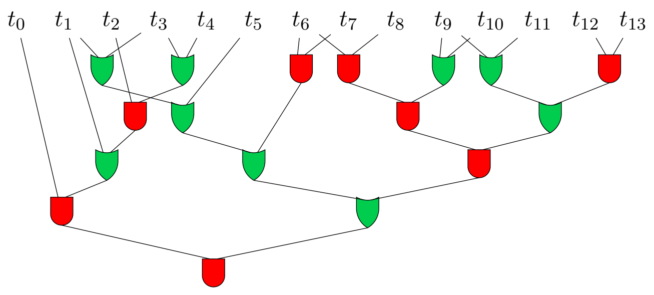
Hence, it remains to bound the number of sets $S_1^{\text{Or}} \subsetneq S^{\text{Or}} \subsetneq I$ considered during the algorithm. For fixed I, S^{Or} and S_1^{Or}, the following holds: A propagate signal of h may by in I or in $T \backslash I$. Each generate signal of h has three options: it is contained in S_1^{Or}, in $S^{\text{Or}} \backslash S_1^{\text{Or}}$ or in $\{t_0, \dots, t_{m-1}\} \backslash S^{\text{Or}}$. Hence, there are at most $3^g 2^p$ partitions for the case that the split gate is an Or.

Similarly, when $\circ =$ And, we have $3^p 2^g$ partitions. Summing up yields the running time bound.

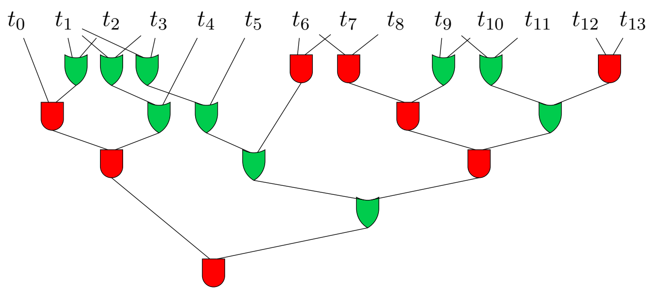
When $h(t;\Gamma)$ is an And-Or path, we have $p, g \in \Big[\lfloor \frac{n}{2} \rfloor, \lceil \frac{n}{2} \rceil\Big]$. Hence, the running time follows from the previous statement. $\square$

We call a circuit C **strongly delay-optimum** if each sub-circuit of C has optimum delay. Note that the formula circuit constructed by our algorithm is strongly delay-optimum. Our algorithm can naturally be adapted to compute a size-optimum circuit among all strongly delay-optimum circuits by storing both delay and size for each generalized And-Or path in line 17 and updating it accordingly. However, for computing a delay-optimum circuit with minimum size among all delay-optimum circuits, we would need to store multiple candidate circuits for each sub-circuit (cf. Section 6.1.4, where this is done for another algorithm) which we did not implement so far.

In Figure 5.4, we show two depth-optimum formula circuits for the And-Or path $g\big((t_0, \dots, t_{14})\big)$. The circuit in Figure 5.4(a) is a circuit with best depth and size 17 computed by Algorithm 6.3, while the circuit in Figure 5.4(b) is size-optimum among all strongly delay-optimum formula circuits, hence a possible output of Algorithm 5.1. Note that in Figure 5.4(a), the left predecessor of the output gate computes an And-



(a) A size-optimum formula circuit for $g(t)$ with size 17.



(b) A size-optimum circuit among all strongly delay-optimum formula circuits for $g(t)$ with size 18.

Figure 5.4: Two formula circuits for the AND-OR path $g(t)$ with $t = (t_0, \dots, t_{13})$ with optimum depth 5. They only differ in the left sub-circuit of the final output.

OR path on 5 inputs with a depth of 4 and a size of 5. In Figure 5.4(b), we instead use an implementation with depth 3 and size 4, which increases the size by 1, but makes the circuit strongly delay-optimum. This can be verified using the lower bound of $\lceil \log_2 n \rceil$ on each sub-circuit with n inputs.

Note that in Figure 5.4(b), Conjecture 5.2.11 is fulfilled. For instance, for the outermost partition, we have $S^{\text{AND}} = \{t_0, t_2, t_4\} \cup \{t_6, t_8, t_{10}, t_{12}, t_{13}\}$.

There are two other exact algorithms for the special case of depth optimization of AND-OR paths. Grinchuk [Gri13] provides an exact algorithm for depth optimization of AND-OR paths, but with a running time of $\Omega(4^m)$, see Section 2.6.4. The theoretical running time of our algorithm for the special case of AND-OR paths coincides with the running time of the formula enumeration algorithm by Hegerfeld [Heg18] for depth optimization, see his Theorem 4.2.16. However, for depth optimization of AND-OR paths, we shall improve our algorithm to obtain a running time of $\mathcal{O}(m2.02^m)$ in Theorem 5.4.6. In his algorithm, Hegerfeld does not directly enumerate formula circuits for AND-OR paths, but so-called rectangle-good protocol trees for Karchmer-Wigderson games (see Karchmer and Wigderson [KW90]) for AND-OR paths, which originate from the area of communication complexity. From these, he derives his formula circuits.

Hegerfeld [Heg18] computes a formula circuit with optimum size among all strongly delay-optimum formula circuits, although he states that he even computes a size-optimum formula circuit among all delay-optimum formula circuits. For instance, for the AND-OR path on 14 inputs, Hegerfeld reports a size of 18 (see Table 5.4), but in Figure 5.4(a), we saw a depth-optimum formula circuit with size 17.

We shall see in Table 5.4 that the practical running times of Hegerfeld's algorithm are much worse than ours. One reason for this is our more efficient practical implementation which we present in Section 5.5. Another reason is that for depth optimization of And-Or paths, the algorithm and its running time can further be improved as in described the following section.

5.4 Improved Algorithm for Depth Optimization

In the case of uniform arrival times, we partition all generalized And-Or paths into so-called *sp*-equivalence classes, where two generalized And-Or paths with signal partitions $P_0 \mathbin{+\!\!+} \ldots \mathbin{+\!\!+} P_c$ and $P'_0 \mathbin{+\!\!+} \ldots \mathbin{+\!\!+} P'_{c'}$ are considered as *sp*-equivalent if and only if $c = c'$ and $|P_b| = |P'_b|$ for all $b \in \{0, \ldots, c\}$. Then, up to renaming of the input variables, any two *sp*-equivalent And-Or paths are either logically equivalent or dual to each other. In both cases, they have the same optimum depth.

Recall that we can identify each generalized And-Or path considered during the algorithm by its inputs. Whenever we compute an optimum circuit for inputs t' during the algorithm, we instead compute an optimum solution for the *sp*-representative $\tilde{t}$. We define $\tilde{t}$ by mapping the inputs of t' to certain inputs in $\{t_0, \ldots, t_{m-1}\}$.

Assume that $t' = \left(t_{i_0}, \ldots, t_{i_j}\right)$ with $0 \leq i_0 < \ldots, < i_j \leq m-1$. We first map t_{i_0} to t_0. If t_{i_0} and t_0 have different gate types, we need to dualize, i.e., change the gate type for all other inputs as well. For all $r = 1, \ldots, j$, we map t_{i_r} to the next input with the appropriate gate type. As always, the last input t_{i_r} has no specified gate type and is mapped to the next free input. For instance, given an And-Or path on inputs $t = (t_0, \ldots, t_{10})$ with gate types $\Gamma = \{\land, \lor, \ldots, \land\}$, the generalized And-Or path on inputs $t' := (t_2, t_5, t_6, t_8, t_9, t_{10})$ (we may omit the gate types here as they can be read off from Γ) is mapped to its *sp*-representative $\tilde{t} := (t_0, t_1, t_2, t_4, t_5, t_6)$.

Furthermore, during partitioning, we avoid generating redundant partitions: For instance, considering again the *sp*-representative $\tilde{t}$ from above, we have $S^{\text{And}} = \{t_0, t_2, t_4, t_6\}$. Here, the partitions $S^{\text{And}} = \{t_0, t_2\} \mathbin{\dot\cup} \{t_4, t_6\}$ and $S^{\text{And}} = \{t_0, t_4\} \mathbin{\dot\cup} \{t_2, t_6\}$ lead to the generalized And-Or paths f_1 on (t_0, t_1, t_2) and f_2 on (t_1, t_4, t_5, t_6), or g_1 on (t_0, t_1, t_4) and g_2 on (t_1, t_2, t_5, t_6), respectively. For both f_1 and g_1, the *sp*-representative is the generalized And-Or path on (t_0, t_1, t_2), and for both f_2 and g_2, it is the generalized And-Or path on (t_0, t_1, t_2, t_3).

More generally, two partitions $S^\circ = S^\circ_1 \mathbin{\dot\cup} S^\circ_2$ and $S^\circ = T^\circ_1 \mathbin{\dot\cup} T^\circ_2$ lead to the same *sp*-representatives if $|S^\circ_1 \cap P_b| = |T^\circ_1 \cap P_b|$ for every $b \in \{0, \ldots, c\}$. Hence, it suffices to consider those S°_1 for which $S^\circ_1 \cap P_b$ is a prefix of P_b for all $b \in \{0, \ldots, c\}$.

We call the procedure to restrict all computations and partitions to *sp*-representatives the **depth normalization**. This leads to an improved analysis of Algorithm 5.1 in the case of depth optimization of And-Or paths.

Theorem 5.4.1. *Let an* And-Or *path $h(t)$ on inputs $t = (t_0, \ldots, t_{m-1})$ with uniform arrival times $a(t_0) = \ldots = a(t_{m-1})$ be given. Assume that Algorithm 5.1 with depth normalization is applied to compute an optimum circuit realizing $h(t)$. Then, the dynamic programming table needed to store the depth of all generalized* And-Or *paths considered during the computation has exactly $F_{m+1} = \mathcal{O}(\varphi^m)$ entries, where F_{m+1} is the $(m+1)$-th Fibonacci number and $\varphi = \frac{1+\sqrt{5}}{2} \approx 1.618$ is the golden ratio.*

Proof. We show that the indicator vectors $(x_0, \ldots, x_{m-1})$ of *sp*-representatives are exactly the 0-1 strings of length m with the following properties:

(i) We have $x_0 = 1$.

(x_{2i}, x_{2i+1})	Allowed values for (x_{2i-2}, x_{2i-1})
$(0,1)$	all but $(1,0)$, $(2,0)$
$(1,0)$	all
$(1,1)$	all
$(2,0)$	all but $(1,0)$
$(2,1)$	all but $(1,0)$

Table 5.2: Extension rules for Q_n.

(ii) Whenever for some $i \in \{2, \dots, m-1\}$, we have $x_i = 1$, then $x_{i-1} \neq 0$ or $x_{i-2} \neq 0$.

(iii) Let $i \in \{0, \dots, m-1\}$ maximum with $x_I = 1$. Then, $i = 0$ or $x_{i-1} = 1$.

For any *sp*-representative, all conditions are fulfilled: The first property is valid as all generalized AND-OR paths considered have at least 1 input and the first input is always mapped to t_0. The second property holds as otherwise, t_{i-2} could have been chosen instead of t_i. The last property holds as the last input is always mapped to the next free position, ignoring gate types.

For the other direction, it is easy to see that for any input vector t' whose indicator vector satisfies the conditions above, the *sp*-representative is again t'.

Denote the set of 0-1 strings of length n with the properties above by G_n. We will inductively show that $|G_n| = F_{n+1}$.

We have $|G_1| = |\{(1)\}| = 1 = F_2$ and $|G_2| = |\{(1,0),(1,1)\}| = 2 = F_3$.

Now, consider $n \geq 3$. We construct G_n by adding prefixes to elements of G_{n-1} and G_{n-2}: By prepending (1) to any element of G_{n-1}, we obtain all elements of G_n that start with $(1,1)$. By prepending $(1,0)$ to elements of G_{n-2}, we obtain all elements of G_n starting with $(1,0,1)$ and, additionally, the invalid element $(1,0,1,0,0,\dots,0)$ which violates the third rule. Moreover, so far, we miss all elements of G_n starting with $(1,0,0)$. But by the second condition, the only such valid element is $(1,0,0,\dots,0)$. Together with the induction hypothesis, we obtain

$$|G_n| = |G_{n-1}| + \big(|G_{n-2}| - 1\big) + 1 = F_n + F_{n-1} = F_{n+1}\,. \qquad \square$$

Using similar, but more involved techniques, we will estimate the running time of Algorithm 5.1 with depth normalization in Theorem 5.4.6. For this, we will prove that the partitions considered in the algorithm essentially correspond to elements of the set Q_n defined as follows.

Definition 5.4.2. Let $Q_1 := \{(0,1),(1,0),(1,1),(2,0),(2,1)\}$, and for $n \in \mathbb{N}_{\geq 2}$, let Q_n be the set of 0-1-2 strings $(x_0, \dots, x_{2n-1})$ of length $2n$ such that for all $i \in \{0, \dots, n-1\}$, the following conditions are fulfilled:

(i) We have $(x_{2i}, x_{2i+1}) \in Q_1$.

(ii) If $i > 0$, the entries $(x_{2i-2}, x_{2i-1}, x_{2i}, x_{2i+1})$ satisfy the extension rules from Table 5.2.

For instance, we have $(2,1,1,0) \in Q_2$ and $(1,0,2,0) \notin Q_2$. Note that the rules imply that there are no consecutive zeroes.

Lemma 5.4.3. *Let $n \in \mathbb{N}_{\geq 1}$. Then, we have $|Q_n| \in \mathcal{O}(\beta_1^n)$, where*

$$\beta_1 = \frac{1}{3}\left(5 + \sqrt[3]{\frac{1}{2}\left(97 - 3\sqrt{69}\right)} + \sqrt[3]{\frac{1}{2}\left(97 + 3\sqrt{69}\right)}\right) < 4.08$$

is the unique real root of the polynomial $\pi(x) := x^3 - 5x^2 + 4x - 1$.

Proof. We will show that for $n \geq 4$, we have

$$|Q_n| = 5|Q_{n-1}| - 4|Q_{n-2}| + |Q_{n-3}|\,. \tag{5.6}$$

From this, the statement can be deduced as follows: The characteristic polynomial $\pi(x)$ of $|Q_n|$ has three distinct roots $\beta_1, \beta_2, \beta_3$ with

$$\beta_1 \approx 4.08\,, \quad \beta_2 \approx 0.46 - 0.18i\,, \quad \beta_3 \approx 0.46 + 0.18i\,.$$

Using Theorem 6.7.8 in Conradie and Goranko [CG15], it follows that there are coefficients $\lambda_1, \lambda_2, \lambda_3 \in \mathbb{C}$ such that for all $n \in \mathbb{N}_{\geq 1}$, we have $|Q_n| = \lambda_1\beta_1^n + \lambda_2\beta_2^n + \lambda_3\beta_3^n$. Since $|\beta_2| = |\beta_3| < \beta_1$, we obtain $|Q_n| \in \mathcal{O}(\beta_1^n)$.

Now, we show Equation (5.6). The idea it to construct Q_n from Q_{n-1}, Q_{n-2}, and Q_{n-3} by adding suffixes.

Let $Q'_n := \{\, a +\!\!+\, b : a \in Q_{n-1}, b \in Q_1 \,\}$. Clearly, $Q_n \subseteq Q'_n$, but not all elements of Q'_n satisfy the extension rules. Let

$$Q''_n = \Big\{ a +\!\!+\, b : a \in Q_{n-2}, b \in \{\, (1,0,0,1), (2,0,0,1), (1,0,2,0), (1,0,2,1) \,\} \Big\}\,.$$

Then, by Table 5.2 we have $Q'_n \backslash Q_n \subseteq Q''_n$. It remains to determine $|Q''_n \backslash (Q'_n \backslash Q_n)|$. As $Q_n \cap Q''_n = \emptyset$, we have $Q''_n \backslash (Q'_n \backslash Q_n) = Q''_n \backslash Q'_n$.

Note that elements in Q'_n fulfill the extension rules from Table 5.2 for all $i \leq n-2$, while elements in Q''_n fulfill the extension rules for all $i \leq n-3$. Hence, an element in $Q''_n \backslash Q'_n$ must violate the extension rule for $i = n-2$. Since no extension rule forbids appending $(1,0)$, all elements in $Q''_n \backslash Q'_n$ end with $(2,0,0,1)$. Furthermore, the only forbidden entry before $(2,0)$ is $(1,0)$. Hence, we have

$$Q''_n \backslash Q'_n \subseteq \{\, a +\!\!+\, (1,0,2,0,0,1) : a \in Q_{n-3} \,\}\,.$$

As before $(1,0)$, any entry is allowed, we even have equality.

Together, we have

$$\begin{aligned}
|Q_n| &= |Q'_n| - |Q'_n \backslash Q_n| \\
&= |Q'_n| - \Big(|Q''_n| - \Big| Q''_n \backslash (Q'_n \backslash Q_n) \Big| \Big) \\
&= |Q'_n| - |Q''_n| + |Q''_n \backslash Q'_n| \\
&= 5|Q_{n-1}| - 4|Q_{n-2}| + |Q_{n-3}|\,.
\end{aligned}$$

This proves Equation (5.6) and thus the lemma. ☐

In order to use the sets Q_n for estimating our running time, we need the following intermediate construction.

Definition 5.4.4. For $n \in \mathbb{N}_{\geq 1}$, let $R_n = \bigcup_{i=1}^{n} \{a +\!\!+\, 0^{2(n-i)} : a \in Q_i\}$ be the set of strings of length $2n$ arising from an element of any Q_i with $i \in \{1, \ldots, n\}$ by appending $2(n-i)$ zeroes.

Observation 5.4.5. Note that by Definition 5.4.4 and Lemma 5.4.3, we have

$$|R_n| = \sum_{i=1}^{n} |Q_i| \in \mathcal{O}\left(\sum_{i=1}^{n} \beta_1^i\right) = \mathcal{O}(\beta_1^n)\,.$$

Theorem 5.4.6. *In the case of Theorem 5.4.1, the running time of Algorithm 5.1 with depth normalization is at most $\mathcal{O}(m\alpha^m)$, where*

$$\alpha := \sqrt{\beta_1} = \sqrt{\frac{1}{3}\left(5 + \sqrt[3]{\frac{1}{2}\left(97 - 3\sqrt{69}\right)} + \sqrt[3]{\frac{1}{2}\left(97 + 3\sqrt{69}\right)}\right)} \leq 2.02$$

and β_1 is defined as in Lemma 5.4.3.

Proof. We count the number of partitions of S° considered by the algorithm for *sp*-representatives. As in Theorem 5.4.1, we encode this situation in strings with certain properties and then estimate the number of these strings.

Consider an *sp*-representative $h(t';\Gamma')$ with input set $I = \{t_{i_0}, \dots, t_{i_{l-1}}\}$ such that $0 \leq i_0 < \dots < i_{l-1} \leq m-1$. Let $I = P_0 + \!\!\!+ \dots + \!\!\!+ P_c$ be the signal partition of $h(t';\Gamma')$, and consider a partition $S^\circ = S_1^\circ \cup S_2^\circ$ of its same-gate inputs. Due to depth normalization, $S_1^\circ \cap P_b$ is a prefix of P_b for all $b \in \{0, \dots, c\}$. Let x denote the 0-1-2 string arising from t by mapping each input that is not contained in I to 0, each input $t_i \in S_1^\circ$ to 2 and each input $t_i \in S_2^\circ$ to 1, and each other input of I to 1.

Note that this is the same proof idea as in Theorem 5.4.1, where we had 3 possible states for the generate signals and 2 possible states for the propagate signals.

We define

$$x' := \begin{cases} x + \!\!\!+ (0) & \text{if } t_0 \in S^\circ, \\ (0) + \!\!\!+ x & \text{otherwise,} \end{cases} \quad \text{and} \quad x'' := \begin{cases} x' & \text{if } m \text{ odd,} \\ x' + \!\!\!+ (0) & \text{otherwise.} \end{cases}$$

Now, x'' has $2n$ entries, where $n := \left\lceil \frac{m+1}{2} \right\rceil$. We will show that $x'' \in R_n$. From this, the result follows: The mapping $(I, S^\circ, S_1^\circ, S_2^\circ) \mapsto x''$ is clearly injective. Hence, $|R_n|$ is an upper bound on the number of partitions considered. Since normalization can be implemented with a running time of $\mathcal{O}(m)$, we obtain a total running time of

$$\mathcal{O}(m|R_n|) = \mathcal{O}(m\beta_1^n) = \mathcal{O}(m\beta_1^{m/2}) = \mathcal{O}(m\alpha^m)\,.$$

Thus, it suffices to prove the following claim.

Claim. We have $x'' \in R_n$.

Proof of claim: All elements of $S^\circ \backslash \{t_{i_{l-1}}\}$ correspond to even entries of x''. Due to depth normalization, x does not contain two consecutive zeroes except for trailing zeroes. As $t_0 \in I$ by normalization and hence $x_0 \neq 0$, the same holds for x' and x''.

Now, it remains to show that for any $i \in \{1, \dots, n-1\}$ with $x_{2i} \neq 0$ or $x_{2i+1} \neq 0$, the extension rules of Table 5.2 are fulfilled for $(x_{2i-2}, x_{2i-1}, x_{2i}, x_{2i+1})$. The case that $(x_{2i-2}, x_{2i-1}, x_{2i}, x_{2i+1}) = (x_{2i-2}, 0, 0, 1)$ with $x_{2i-2} \neq 0$ is already excluded as there must not be consecutive zeroes before a 1. The case $(x_{2i-2}, x_{2i-1}, x_{2i}, x_{2i+1}) = (1, 0, 2, x_{2i+1})$ with x_{2i+1} arbitrary cannot occur as here, the entries in t' corresponding to x_{2i-2} and x_{2i} are in the same input group P_b' for some $b \in \{0, \dots, c\}$, hence, by normalization, we have $x_{2i-2} \geq x_{2i}$. All other configurations are permitted.

Hence, we have $x'' \in R_n$. □

This proves the theorem. □

Apparently, the sequence $(|Q_n|)_{n \in \mathbb{N}}$ is given by sequence A012814 in the OEIS [Slo], which consists of every 5th entry of the **Padovan sequence**, see sequence A000931 in the OEIS. The growth rate of the Padovan sequence is given by $\rho := \sqrt[5]{\beta_1}$ which is also known as the **plastic number**. Hence, the running time of our algorithm can also be expressed as $\mathcal{O}\left(m\left(\rho^{5/2}\right)^m\right)$.

5.5 Practical Implementation

In this section, we assume that all input arrival times are integral as this allows more efficient speed-up techniques and exact integer arithmetic. When fractional input arrival times are given, we use the binary-search extension from Theorem 5.1.5 to reduce the problem to several integral instances with a running time increase of at most a factor of $\mathcal{O}(\log_2 m)$.

We implemented Algorithm 5.1 in a C++ program, using 64-bit bit sets to encode the generalized And-Or path instances via the bijection κ to subsets of $\{t_0, \dots, t_{m-1}\}$. In order to obtain good practical running times, we implemented several speed-up techniques. On most instances, these in particular imply that we only compute the delay for only a fraction of the generalized And-Or paths from our dynamic programming table, see Table 5.5. Hence, we store the table in a hash set, which violates the worst-case running time guarantee of Algorithm 5.1, but is much faster in practice and, more important, much less memory-consuming.

For describing our speed-up techniques, assume that we apply Algorithm 5.1 to a generalized And-Or path $h(t;\Gamma)$ with m inputs and arrival times $a(t_0), \dots, a(t_{m-1})$. Moreover, when the procedure `compute_opt` is applied to a subset $I \subseteq \{t_0, \dots, t_{m-1}\}$ with $I = \left\{t_{i_0}, \dots, t_{i_{r-1}}\right\}$ for $0 \leq i_0 < \ldots < i_{r-1} \leq m-1$, we denote the corresponding generalized And-Or path by $h(t';\Gamma')$ and its signal partition by $P_0 \mathbin{+\!\!\!+} \ldots \mathbin{+\!\!\!+} P_c$.

When we apply our algorithm for depth optimization, we use the depth normalization as in Theorem 5.4.1. Most of the other speed-ups techniques are based on lower bounds and upper bounds on the delay. For a generalized And-Or path $h(t';\Gamma')$, we maintain not only the best delay of a circuit for $h(t';\Gamma')$ computed so far (and, in the size-optimization mode, the best possible size for the best possible delay), but also a lower bound on its delay. Furthermore, when calling the procedure `compute_opt`, we assume that we are given a parameter D and are supposed to find a circuit with best delay among all solutions with delay at most D. For the two sub-functions considered in partitioning, we hence may use an upper bound of $D-1$ when computing their table entries.

Now, it is possible that we do not find a solution when applying the procedure `compute_opt`. In this case, we may update the lower bound to $D+1$. On the other hand, if during partitioning, we find a solution with delay $d \leq D$, in case of the non-size-optimization mode, we are only interested in another solution if it has delay strictly smaller than d, and in case of the size-optimization mode, if it has delay at most d. Hence, we may update the upper bound D to $d-1$ or d in the respective mode for the remaining partitions to be considered. Note that if we allowed fractional arrival times, we would not be able to subtract 1 here.

For the outermost call of the algorithm, we set $D = \infty$. We call the mechanism that handles upper bounds during the algorithm **upper bound propagation**.

Having very good lower and upper bounds has a high impact on the running time, so we carefully use any information available to update our bounds.

Assume now that we apply `compute_opt` to compute a table entry, i.e., to find an optimum circuit for the generalized AND-OR path $h(t';\Gamma')$ with input set I with delay at most D. Before starting our partitioning process (see Section 5.5.2), we compute several lower bounds as in the following section. If any of these is larger than D, we know that there is no circuit with delay at most D for $h(t';\Gamma')$ and need not start the partitioning process.

5.5.1 Lower Bounds

A **basic lower bound** that can be computed quickly for any generalized AND-OR path $h(t';\Gamma')$ arises from the lower bounds in Theorem 2.3.15 and Corollary 5.2.5, i.e.,

$$\max\left\{ \lceil \log_2 W(t') \rceil, \max\left\{ \max_{t_i \in P_0} a(t_i) + 1, \max_{t_i \in P_b : b > 0} a(t_i) + 2 \right\} \right\},$$

where $W(t') = \sum_{j=i_0}^{i_{r-1}-1} 2^{a(t_{i_j})}$ as in Definition 2.3.16. as in Definition 2.5.6. Note that the first lower bound requires integral arrival times.

We use two other **reducing lower bounds** that each consider a specific reduced generalized AND-OR path $h(t'';\Gamma'')$ of $h(t';\Gamma')$ with similar structural complexity. For $h(t'';\Gamma'')$, we recursively apply the algorithm with depth bound D. Either there is no solution, in which case $D+1$ is a lower bound on the optimum delay for $h(t'';\Gamma'')$, thus also for $h(t';\Gamma')$; otherwise, we know the optimum delay for $h(t'';\Gamma'')$, which is a lower bound for $h(t';\Gamma')$. This usually yields a strong lower bound, but is very time-consuming.

First, only in the special case of depth optimization, we consider the generalized AND-OR path $h(t'';\Gamma'')$ arising from $h(t';\Gamma')$ by keeping only the largest input group in the signal partition completely and condensing each other input group to a single input (except for the last group, which keeps 2 inputs). In the case of depth optimization, only the input-group sizes matter, so there are only $\mathcal{O}(m^3)$ of these generalized AND-OR paths, and it is not harmful to solve them optimally.

Secondly, also in the case of delay optimization, we consider a reduced generalized AND-OR path $h(t'';\Gamma'')$ that arises from removing a single input of $h(t';\Gamma')$ in a way that hopefully the optimum delay of any circuit for $h(t'';\Gamma'')$ is the same as for $h(t';\Gamma')$. Hence, among all inputs with the minimum arrival time, we remove an input of the largest input group. Empirically, we see that in the case of depth optimization, this lower bound is tight in 97% of its applications. This matches the observation that if we iteratively apply this lower bound m times, starting with a generalized AND-OR path with optimum depth d, the optimum depth changes only d times, where $d \ll m$.

5.5.2 Partitioning the Same-Gate Inputs

For determining a solution with delay D for a generalized AND-OR path $h(t';\Gamma')$ – if it exists –, we enumerate partitions $S^\circ = S_1^\circ \cup S_2^\circ$ of its same-gate input set S° for all $\circ \in \{\text{AND}, \text{OR}\}$ in line 13 of Algorithm 5.1. In our implementation, we first choose $\circ := \circ_0$ as the gate type of the input group P_0 as empirically, this more often yields a good circuit, and afterwards the other gate type. For both, we enumerate partitions of S° and recursively try to find a solution with delay at most D.

We avoid generating too many partitions of a set S° by enumerating the partitions in a specific order. In a recursive approach, one by one, we assign the inputs to one of the subsets of S°. Here, just as in standard branch-and-bound algorithms, we follow the idea to make the most important decisions first. Recall from the proof of Theorem 5.3.1 that by convention, the last input $t_{i_{r-1}}$ is always contained in S_2°.

Now, we first enumerate the highest input index i_l for which input t_{i_l} goes in to the other part, S_1°. Once t_{i_l} is fixed, we have completely determined which of the inputs with different gate type than $\circ$ are contained in in both $h(t';\Gamma')_{S_1^\circ}$ and $h(t';\Gamma')_{S_2^\circ}$, or only in $h(t';\Gamma')_{S_2^\circ}$. Based on this, we compute another lower bound, the **cross-partition Huffman bound**, by Huffman coding on all inputs of $h(t';\Gamma')$, where those inputs that are contained in both sub-functions are counted twice, and may stop when this lower bound exceeds D.

As t_{i_l} is the input with the highest index in S_1°, we already know that all inputs $t_i \in S^\circ$ with $i > i_l$ must be in S_2°. It remains to enumerate those $t_i \in S^\circ$ with $i < i_l$. They are assigned to the sets S_1° and S_2° recursively, in the order of decreasing arrival time, and in case of ties, inputs with larger indices are considered first. For each input, we first put it into S_2° and recursively continue with the other inputs; and then put it into S_1° and go into recursion. This way, we in particular prioritize the construction of consecutive sets S_1° and S_2°, which often allows finding an optimum solution quickly (cf. Section 6.3).

Now, assume that we try to compute a circuit for $h(t';\Gamma')$ with delay at most D via a fixed partition $S^\circ = S_1^\circ \mathbin{\dot\cup} S_2^\circ$. Before computing a solution, we evaluate all lower bounds available for the two sub-instances, and stop if any of the lower bounds is larger than $D-1$. Otherwise, we recursively compute the table entries of $h(t';\Gamma')_{S_1^\circ}$ and $h(t';\Gamma')_{S_2^\circ}$ with delay bound $D-1$. As already mentioned, based on whether we did find a solution or not, we may update the lower bound for $h(t';\Gamma')$.

Note that the lower bound L on the best delay achievable for $h(t';\Gamma')$ is also a lower bound for all generalized And-Or path on a superset of the inputs I of $h(t';\Gamma')$. Hence, if we have updated L for $h(t';\Gamma')$, in **lower bound propagation**, we also update the lower bound for certain generalized And-Or paths whose inputs are a superset of I. Doing this for all supersets would be to costly; so we only update lower bounds of supersets which are already contained in our dynamic programming table and arise from adding a single input. For those whose lower bounds are improved, we recurisvely repeat this procedure.

If we did not find a solution with delay at most D for the current partition, we might discard a part of our enumeration tree in **subset enumeration pruning**: Consider the inputs of S° in the order $t_{j_0}, \dots, t_{j_l}$ in which we enumerate whether to put them into S_1° or S_2°. When considering an input t_{j_i}, we have already assigned the inputs $t_{j_0}, \dots, t_{j_{i-1}}$ to one of the two subsets. If we add t_{j_i} to S_2°, the set S_2° is minimal among all sets that will arise from enumerating assignments for the elements $t_{j_{i+1}}, \dots, t_{j_l}$. The first assignment that will be tried for $t_{j_{i+1}}, \dots, t_{j_l}$ is to put them all into S_1°. Hence, when the computation of a solution with delay at most D for this generalized And-Or path was not successful because the And-Or path $h(t';\Gamma')_{S_2^\circ}$ had too large delay, we already know that all other partitions with $t_{j_0}, \dots, t_{j_i}$ unchanged will also not lead to delay at most D. Hence, we can skip this part of our enumeration tree. The same holds when adding t_{j_i} to S_1°.

Finally, we note that the running time for the computation of a table entry highly depends on D. Hence, when computing a table entry with a lower bound of L, in **delay probing**, we in fact loop over all possible delays $d \in \{L, \dots, D\}$ with increasing d and try to find a solution with delay d. The first value d for which a solution is found is then the optimum delay of any circuit for $h(t';\Gamma')$.

	Integral arrival times		Fractional arrival times	
# inputs	With size opt.	No size opt.	With size opt.	No size opt.
10	0.001	0.000	0.005	0.000
20	0.442	0.002	2.248	0.008
30	2374.234	0.012	5790.565	0.092
40	-	0.096	-	44.388
50	-	8.294	-	8.223
60	-	3.514	-	*106.860

Table 5.3: Average running times of Algorithm 5.1 on 10 randomly generated And-Or path instances for each number of inputs. For fractional arrival times and the non-size-opt mode, we omit one instance with 60 inputs because there, the memory limit of 400 GB was reached and the run could not finish.

5.6 Computational Results

In Section 5.6.1, we examine the empirical running time of our algorithm, in particular the quality of our speed-up techniques. We will see that for up to 64 inputs, we can compute the optimum depth of any And-Or path circuit. From this, together with a theoretical result, we will derive the optimum depth of n-bit adder circuits for n that are a power of two for up to $n = 8192$ bits in Section 5.6.2.

5.6.1 Running Time Comparisons

In Table 5.3, we state the average running single-threaded times of our algorithm on delay-optimization And-Or path instances on two testbeds: one with integral, one with fractional arrival times. For each of these testbeds and each number $n \in \{10, 20, 30, 40, 50, 60\}$ of inputs, we created 10 instances with random arrival times from the interval $[0, n]$. Later, we use the same testbed for the running time analysis of Algorithm 6.3 in Table 6.2.

On both testbeds, we show the running times of our algorithm executed single-threadedly on a machine with two Intel(R) Xeon(R) CPU E5-2699 v4 processors, both for the computation of the optimum delay ("No size opt."), and of the optimum delay and optimum size of a strongly delay-optimum formula circuit ("With size opt."). We see that running times are much higher in the size optimization mode. Here, solving the first instance of the integral testbed with 40 inputs already took 18 hours, so we cannot display the average running times for more than 30 inputs. Without size optimization, almost any instance can be solved in a few seconds, only for one instance with 60 inputs, our memory limit of 400 GB was reached. This run is not counted in the statistics. The effectiveness of our pruning strategies varies drastically depending on the arrival time profile, thus also our running times. For instance, for the integral runs with size optimization, the running times on instances with 30 inputs vary from 35 up to 14969 seconds. By examining instances with a high running time, we could most likely further improve our speed-up techniques.

Now, we consider the And-Or path Circuit Depth Optimization Problem. Note that up to duality, for this, there is exactly one instance for a fixed number of inputs. In Table 5.4, we give a comparison of Algorithm 5.1 with the formula enumeration algorithm by Hegerfeld [Heg18]. On any instance solved both by Hegerfeld's algorithm and our algorithm, the computed optimum depths coincide; and using our size-optimization mode, we verify that Hegerfeld computes the optimum size of any strongly depth-optimum circuit on each instance.

Hegerfeld's running times are taken from [Heg18]; our runs were executed on a machine with two Intel(R) Xeon(R) CPU E5-2687W v3 processors. All of Hegerfeld's runs with up to 27 inputs are also run with a single thread, but the runs for 28 and 29 inputs were executed in parallel and the stated running time is the wall time multiplied by the number of threads. For our algorithm, we again show running times for the computation of the optimum depth and optimum size of a strongly depth-optimum formula circuit ("With size opt."), and for the computation of the optimum depth only ("No size opt."). Hegerfeld always computes a size-optimum strongly depth-optimum formula circuit.

For up to 14 inputs, Hegerfeld's algorithm runs less than a second. The largest instance he can solve has 29 inputs.

# Inputs	Depth	Size	[Heg18] [s]	Algorithm 5.1 [s]	
			With size opt.	With size opt.	No size opt.
1	0	0	0	0.000	0.000
2	1	1	0	0.000	0.000
3	2	2	0	0.000	0.000
4	3	3	0	0.000	0.000
5	3	5	0	0.000	0.000
6	3	6	0	0.000	0.000
7	4	7	0	0.000	0.000
8	4	9	0	0.000	0.000
9	4	10	0	0.000	0.000
10	4	13	0	0.000	0.000
11	5	13	0	0.001	0.000
12	5	14	0	0.002	0.000
13	5	16	0	0.004	0.000
14	5	18	0	0.005	0.000
15	5	20	1	0.007	0.000
16	5	21	2	0.008	0.000
17	5	24	4	0.008	0.000
18	5	25	11	0.009	0.000
19	5	29	27	0.015	0.002
20	6	27	71	0.234	0.005
21	6	28	180	0.358	0.007
22	6	31	463	0.588	0.008
23	6	32	1035	0.923	0.008
24	6	35	2893	1.259	0.007
25	6	36	7214	1.631	0.007
26	6	38	22661	2.097	0.007
27	6	40	60598	2.401	0.007

# Inputs	Depth	Size	[Heg18] [s]	Algorithm 5.1 [s]	
			With size opt.	With size opt.	No size opt.
28	6	42	$\leq$ 480960	2.680	0.007
29	6	44	$\leq$ 2775000	2.763	0.007
30	6	47		2.927	0.008
31	6	49		2.991	0.008
32	6	53		3.068	0.009
33	6	57		3.159	0.010
34	7	51		1822	0.300
35	7	53		2921	0.861
36	7	55		5145	0.978
37	7	57		8064	0.958
38	7	59		13949	0.961
39	7	61		19539	0.957
40	7	63		33778	0.974
41	7	65		53287	0.954
42	7	67		87514	0.945
43	7	70		143409	0.939
44	7	$\leq$73			0.945
45	7	$\leq$76			0.958
46	7	$\leq$77			0.941
47	7	$\leq$83			1.285
48	7	$\leq$84			1.406
49	7	$\leq$84			1.399
50	7	$\leq$85			1.404
51	7	$\leq$89			1.410
52	7	$\leq$90			1.405
53	7	$\leq$93			1.407
54	7	$\leq$94			1.409
55	7	$\leq$98			1.415
56	7	$\leq$99			1.410
57	7	$\leq$104			1.406
58	7	$\leq$105			1.395
59	7	$\leq$109			1.413
60	7	$\leq$110			1.425
61	8	$\leq$ 111			4574
62	8	$\leq$ 113			8468
63	8	$\leq$ 114			9729
64	8	$\leq$ 117			9037

Table 5.4: Running times of Algorithm 5.1 compared with the enumeration algorithm by Hegerfeld [Heg18]. All reported running times are single-threaded. For 28 and 29 inputs, Hegerfeld ran his algorithm in parallel; here, the reported running times are the wall time multiplied by the number of threads. Dashed lines separate instances with the same optimum depth.

Our algorithm can solve all instances with up to 64 inputs. Note that our implementation uses 64-bit bit sets to encode the generalized And-Or path instances, hence, we currently cannot consider larger instances. By adjusting the bit sets used, this technicality can be overcome. However, we do not expect to solve an instance with 110 inputs, where the next change in depth is likely, see Table 5.6.

In order to examine the quality of our various speed-up techniques, we define 5 scenarios: in scenario 1, we run the basic algorithm without any enhancements; in scenario 5, we enable all speed-up techniques from Section 5.5. The intermediate scenarios all add a selection of speed-ups to the previous scenario:

- Scenario 1: No speed-ups.
- Scenario 2: Add depth normalization.
- Scenario 3: Add upper bound propagation, basic lower bound.
- Scenario 4: Add cross-partition Huffman bound, subset enumeration pruning.
- Scenario 5: Add reducing lower bounds, lower bound propagation, delay probing.

For each scenario, we ran the algorithm with all depth optimization instances with at least 21 inputs – we only state results for an instance-scenario pair if the running time is at most 8 hours. For each run, we store the number E of table entries for which the partitioning process has been started at least once and the number P of partitions considered. In Table 5.5, we show the logarithms of these numbers, rounded to the nearest integer, and the running times.

In general, for fixed m, the number of entries and partitions and the running time reduces significantly with increasing scenario number. From scenario 3 on, we can solve the instance with 34 inputs within the running time limit of 8 hours, which is the first instance with an optimum depth of 7. Using all pruning techniques in scenario 5, we can solve any instance with up to 64 inputs within 3 hours. In particular, note that in contrast to scenarios 1 - 4, in scenario 5, the running time does not necessarily increase with increasing m. In a range of inputs where the optimum depth does not increase (e.g., from 34 up to 60 inputs), our reducing lower bounds have a high impact.

Note that, as estimated by Theorem 5.3.1, for each number m of inputs, for scenario 1, we have $E \approx 2^m$, and the running time increases by a factor of roughly $\sqrt{6}$ when m increases by 1. For scenario 2, we have checked that – as proven in Theorem 5.4.1 – the exact number of entries for m inputs is exactly the Fibonacci number F_{m+1}. Note that from m to $m+1$, the running time grows roughly by a factor of $\alpha = 2.02$, which matches the running time guarantee shown in Theorem 5.4.6.

5.6.2 Optimum Depths of Adder Circuits

From Table 5.4, we now know depth-optimum And-Or path circuits for up to 64 inputs. Furthermore, based on Corollary 5.2.10, given $d \in \mathbb{N}$, we can deduce an upper bound on the maximum number $m = m(d, 0)$ for which an And-Or path on m inputs can be realized by a circuit with depth d.

For example, our results from Table 5.4 directly yield $m(7, 0) = 60$. By Corollary 5.2.10, we have $m(8, 0) \leq 2 \cdot 60 = 120$. Applying this corollary again, we obtain $m(9, 0) \leq 2m(8, 0) \leq 240$. Iterative application of Corollary 5.2.10 hence yields the

	Scenario 1			Scenario 2			Scenario 3			Scenario 4			Scenario 5		
m	$\log_2 E$	$\log_2 P$	T [s]	$\log_2 E$	$\log_2 P$	T [s]	$\log_2 E$	$\log_2 P$	T [s]	$\log_2 E$	$\log_2 P$	T [s]	$\log_2 E$	$\log_2 P$	T [s]
21	21	29	35.4	14	23	1.6	10	19	0.1	10	15	0.0	9	13	0.0
22	22	30	94.6	15	24	3.5	11	19	0.1	10	16	0.0	9	14	0.0
23	23	31	237.9	16	25	6.9	11	20	0.2	11	16	0.0	9	14	0.0
24	24	32	630.6	16	26	15.4	11	20	0.2	11	16	0.0	9	14	0.0
25	25	34	1540.4	17	27	32.1	11	20	0.2	11	16	0.0	9	14	0.0
26	26	35	4055.7	18	28	69.7	11	20	0.2	11	16	0.0	9	14	0.0
27	27	36	10034.2	18	29	142.4	11	21	0.3	11	17	0.0	9	14	0.0
28	28	38	25055.1	19	30	315.9	11	21	0.4	11	17	0.0	9	14	0.0
29				20	31	642.0	12	22	0.8	11	17	0.0	9	14	0.0
30				20	32	1406.2	12	23	1.4	11	17	0.0	9	14	0.0
31				21	33	2939.7	12	24	2.6	11	17	0.0	9	14	0.0
32				22	34	6445.6	12	25	6.8	11	17	0.0	9	14	0.0
33				22	35	13062.4	12	26	14.3	11	17	0.0	9	14	0.0
34							17	31	623.0	16	26	14.1	11	19	0.3
35							18	33	1666.7	17	27	34.2	12	21	0.9
36							19	33	3066.7	18	27	45.4	12	21	1.0
37							19	34	6013.4	18	28	60.9	12	21	1.0
38							20	35	9211.6	19	28	75.8	12	21	1.0
39							20	36	15861.5	19	28	92.2	12	21	1.0
40							21	36	22140.3	19	29	109.8	12	21	1.0
41										20	29	135.7	12	21	1.0
42										20	29	145.1	12	21	0.9
...											...			...	
59										21	29	224.7	12	22	1.4
60										21	29	226.6	12	22	1.4
61													18	32	4574.9
62													18	33	8468.1
63													18	33	9729.8
64													18	33	9037.3

Table 5.5: Comparison of speed-up scenarios. The number of table entries computed is denoted by E, the number of partitions computed by P, and the running time in seconds by T. Dashed lines separate ranges of instances with the same optimum depth.

d	Lower bound [Gri13]	Old upper bound [Heg18] and Cor. 5.2.10	New upper bound Alg. 5.1 and Cor. 5.2.10
0	1	1	1
1	2	2	2
2	3	3	3
3	6	6	6
4	10	10	10
5	19	19	19
6	33	38	33
7	60	76	60
8	109	152	120
9	202	304	240
10	375	608	480
11	698	1216	960
12	1311	2432	1920
13	2466	4864	3840
14	4645	9728	7680
15	8782	19456	15360
16	16627	38912	30720
17	31548	77824	61440
18	60059	155648	122880

Table 5.6: Lower and upper bounds on $m(d, 0)$ for $d \in \{0, \dots, 18\}$.

upper bounds on $m(d, 0)$ for small d as shown in the third column of Table 5.6. Applying the same with the results by Hegerfeld [Heg18] yields the second column.

In the first column of Table 5.6, we show the best available lower bounds on $m(d, 0)$ as computed by the heuristics from Grinchuk [Gri13], i.e., for each d in the table, we report the maximum value m for which an And-Or path circuit with depth d is known.

Comparing the first and the third column, we directly see that up to 60 inputs, the circuits from [Gri13] are optimum. But $m(7, 0) \leq 60$ implies that for $m \geq 61$, a depth of at least 8 is needed, so Grinchuk's circuits are actually optimum for up to 109 inputs. Moreover, $m(8, 0) \leq 120$ implies that for $121 \leq m \leq 202$, Grinchuk's circuits also have optimum depth, and so on. The ranges of inputs m for which we know an optimum And-Or path realization resulting from this are shown in Table 5.7. Here, in the left column, we show the ranges of inputs for which an optimum solution is known as derived from the results of Hegerfeld [Heg18], Grinchuk [Gri13], and Corollary 5.2.10; and in the right column, we exchange Hegerfeld's results by ours.

Recall from Equation (2.18) that the final carry c_n bit of an n-bit adder is an And-Or path on $2n - 1$ inputs and that – when circuit size and fanout are not regarded – optimum adder circuits on n bits yield optimum And-Or path circuits on $2n - 1$ inputs, and vice versa. Hence, in particular, Tables 5.4 and 5.7 yield the optimum depths of all adder circuits with 2^k inputs for $k \leq 13$. We show these in Table 5.8.

d	[Gri13],[Heg18],		Cor. 5.2.10	[Gri13],	Alg. 5.1,	Cor. 5.2.10
0	1	$\leq m \leq$	1	1	$\leq m \leq$	1
1	2	$\leq m \leq$	2	2	$\leq m \leq$	2
2	3	$\leq m \leq$	3	3	$\leq m \leq$	3
3	4	$\leq m \leq$	6	4	$\leq m \leq$	6
4	7	$\leq m \leq$	10	7	$\leq m \leq$	10
5	11	$\leq m \leq$	19	11	$\leq m \leq$	19
6	20	$\leq m \leq$	33	20	$\leq m \leq$	33
7	39	$\leq m \leq$	60	34	$\leq m \leq$	60
8	77	$\leq m \leq$	109	61	$\leq m \leq$	109
9	153	$\leq m \leq$	202	121	$\leq m \leq$	202
10	305	$\leq m \leq$	375	241	$\leq m \leq$	375
11	609	$\leq m \leq$	698	481	$\leq m \leq$	698
12	1217	$\leq m \leq$	1311	961	$\leq m \leq$	1311
13	2433	$\leq m \leq$	2466	1921	$\leq m \leq$	2466
14				3841	$\leq m \leq$	4645
15				7681	$\leq m \leq$	8782
16				15361	$\leq m \leq$	16627
17				30721	$\leq m \leq$	31548

Table 5.7: Numbers m of inputs for which we can show that the optimum depth of an AND-OR path circuit on m inputs is d.

n	$2n-1$	d
1	1	0
2	3	2
4	7	4
8	15	5
16	31	6
32	63	8
64	127	9
128	255	10
256	511	11
512	1023	12
1024	2047	13
2048	4095	14
4096	8191	15
8192	16383	16

Table 5.8: Optimum depths of adder circuits for n input pairs, where n is a power of 2. The middle column shows the number of inputs of the AND-OR path computing the most significant carry bit.

Chapter 6

And-Or Path Optimization in Practice

In this chapter, we develop a delay optimization algorithm for And-Or paths that is used for logic optimization in the IBM VLSI design flow, see also Chapter 7.

Our exact delay optimization algorithm from Chapter 5 is not suitable for this purpose: In practice, circuit size as an important secondary criterion. Our exact algorithm computes only strongly delay-optimum formula circuits, which is a disadvantage for size. Moreover, already our current size optimization mode has too high average running times (e.g., 2.2 seconds for 20 inputs, see Table 5.3), with significant outliers (e.g., more than 9 hours for one instance with 30 inputs).

Instead, we now describe a polynomial-time algorithm which computes very good solutions in practice, as we will demonstrate. The key idea of our algorithm is to compute the best possible circuit over $\Omega_{\text{mon}} = \{\text{And2}, \text{Or2}\}$ resulting from a recursive application of the recursion strategies described in Section 2.6.2 in a dynamic program. The algorithm has been published in concise form in Brenner and Hermann [BH20].

In Section 6.1, we describe and analyze our algorithm, which has a running time of $\mathcal{O}(m^4)$ and allows an effective size reduction technique. We will show that our algorithm fulfills the best known asymptotic delay guarantee of Theorem 4.2.4, see Theorem 6.1.14.

In Section 6.3, we explain the differences of our algorithm to the exact algorithm from Chapter 5. We conjecture that in the special case of uniform arrival times, our dynamic programming algorithm always computes optimum solutions.

In Section 6.2, we show experimental results: On a testbed containing 25000 And-Or path instances with 4 up to 28 inputs, we will demonstrate that our algorithm from Section 6.1 yields significantly better results on small instances compared to the previously best implemented algorithms, i.e., the methods of Held and Spirkl [HS17b] and Rautenbach, Szegedy, and Werber [RSW06]. Even more, we will show that the delays of our circuits are now much closer to the optimum delays computed by the exact algorithm from Chapter 5. We find delay-optimum solutions on more than 95% of the considered instances with integral arrival times, the average difference from the optimum delay is roughly 0.04 and the maximum difference 1. For the best delay among the circuits computed by Held and Spirkl [HS17b] and Rautenbach, Szegedy, and Werber [RSW06], the average difference to the optimum is 1.64, the maximum difference is 4, and only 10% of their circuits are delay-optimum.

6.1 Dynamic Program for Delay Optimization

As in Chapter 4, we in fact present an algorithm for the optimization of extended AND-OR paths (cf. Definition 2.6.14), not only for AND-OR paths. This is motivated by the fact that extended AND-OR paths allow a more flexible recursive circuit construction, see Section 2.6.2. Hence, let symmetric inputs $s = (s_0, \dots, s_{n-1})$ and alternating inputs $t = (t_0, \dots, t_{m-1})$ be given. We are interested in computing circuits for the extended AND-OR paths $f(s,t)$ and $f^*(s,t)$. Using the notation from Section 2.6.2, there are three recursive ways to compute $f(s,t)$ from circuits for smaller extended AND-OR paths which have been introduced in Corollary 2.6.17, Corollary 2.6.18 and Observation 2.6.20:

$$f(s,t) = f\big(s,t'\big) \wedge f^*\Big(\widehat{t'}, t''\Big) \quad \text{for a prefix } t' \text{ of } t \text{ with } |t'| < m \text{ odd} \tag{6.1}$$

$$f(s,t) = f\big(s,t'\big) \vee f\Big(s \mathbin{+\!\!+} \widehat{t'}, t''\Big) \quad \text{for a prefix } t' \text{ of } t \text{ with } |t'| < m \text{ even} \tag{6.2}$$

$$f(s,t) = \operatorname{sym}\big((s_0, \dots, s_{k-1})\big) \wedge f\big((s_k, \dots, s_{n-1}), t\big) \quad \text{for } k < n \tag{6.3}$$

By applying any of these splits, we reduce the problem of optimizing an extended AND-OR path $f(s,t)$ to two problems on instances with strictly fewer inputs: Either, we split off a prefix of the alternating inputs and construct a circuit for $f(s,t)$ based on circuits for two smaller extended AND-OR paths (cf. Equations (6.1) and (6.2)); or we split off a prefix of the symmetric inputs, for which we build an optimum binary tree, construct an extended AND-OR path on the remaining symmetric inputs and t, and combine these two circuits to a circuit for $f(s,t)$ (cf. Equation (6.3)).

Recall that by Corollary 2.5.3, a circuit for $f^*(s,t)$ can be obtained from a circuit for $f(s,t)$ by dualization.

Our algorithm is a dynamic program that, given s and t, computes a delay-optimum circuit for $f(s,t)$ that can be obtained using the recursion formulas Equations (6.1) to (6.3). Regarding instances with integral arrival times, as a preparation, we present a straight-forward algorithm in Section 6.1.1 and our final algorithm in Section 6.1.2. An extension of both algorithms to fractional arrival times is shown in Section 6.1.3; and in Section 6.1.4, we describe how we heuristically improve the number of gates of our circuits.

As a common preparation, we now introduce notation allowing to describe our algorithms in a dynamic-programming fashion.

Notation 6.1.1. Given Boolean input variables $t = (t_0, \dots, t_{m-1})$ and indices $i, j, k \in \{0, \dots, m-1\}$ with $i \leq j \leq k$ and $j - i$ even, we write

$$f_{i,j,k} = f\Big(\big(t_i, t_{i+2}, \dots, t_{j-4}, t_{j-2}\big), \big(t_j, \dots, t_k\big)\Big)$$

and

$$f^*_{i,j,k} = f^*\Big(\big(t_i, t_{i+2}, \dots, t_{j-4}, t_{j-2}\big), \big(t_j, \dots, t_k\big)\Big)\,.$$

We denote the number of inputs of $f_{i,j,k}$ by $N(f_{i,j,k}) \in \mathbb{N}$. Note that

$$N(f_{i,j,k}) = \frac{j-i}{2} + k - j + 1\,.$$

In other words, the functions $f_{i,j,k}$ are extended AND-OR paths on exactly those subsets of the inputs $t_0, \dots, t_{m-1}$ that have a consecutive range of alternating inputs and, preceding this range, symmetric inputs that contain every second input. Note that in particular $f_{0,0,m-1} = g(t)$ is the And-Or path on all inputs t. We can now rewrite the three splits (6.1) to (6.3) using this notation. For odd prefix length $l = 2\lambda + 1 \in \mathbb{N}$ with $1 \leq l \leq k - j$ and thus $\lambda \in \mathbb{N}$, $0 \leq \lambda \leq \frac{k-j-1}{2}$, we have

$$f_{i,j,k} = f_{i,j,j+2\lambda} \wedge f^*_{j+1,j+2\lambda+1,k}\,, \tag{6.4}$$

for even prefix length $l = 2\lambda$ with $2 \leq l \leq k - j$ and thus $\lambda \in \mathbb{N}$, $1 \leq \lambda \leq \frac{k-j}{2}$, we have

$$f_{i,j,k} = f_{i,j,j+2\lambda-1} \vee f_{i,j+2\lambda,k}\,, \tag{6.5}$$

and for $1 \leq \lambda \leq \frac{j-i}{2}$, we have

$$f_{i,j,k} = f_{i,i+2\lambda-2,i+2\lambda-2} \wedge f_{i+2\lambda,j,k}\,. \tag{6.6}$$

Note that once $j - i$ is even, in each of these splits, the difference of the "j" and "i" indices is even for any occurring sub-function. Thus, every split in (6.1) to (6.3) can be represented using the functions $f_{i,j,k}$ for indices i, j, k with $j - i$ even as defined in Notation 6.1.1. Furthermore, note that indeed, each sub-formula occurring in (6.4) to (6.6) has strictly fewer inputs that $f_{i,j,k}$.

6.1.1 Binary-Circuit Dynamic Program

In this section, we formulate a first dynamic program for And-Or path optimization, the **binary-circuit dynamic program**. In the next section, we will see a motivation for this name. Algorithm 6.1 states our algorithm, which works as follows.

Let input variables $t = (t_0, \dots, t_{m-1})$ with arrival times $a(t_0), \dots, a(t_{m-1}) \in \mathbb{N}$ be given. For every $0 \leq i \leq j \leq k \leq m - 1$ with $j - i$ even, Algorithm 6.1 computes a circuit $C_{i,j,k}$ for $f_{i,j,k}$ in the respective iteration of the loop in lines 2 to 7:

In line 4, we compute $C_{i,j,k}$ whenever $f_{i,j,k}$ is a symmetric function, i.e., if $k \in \{j, j + 1\}$. Here, $C_{i,j,k}$ is a delay-optimum circuit computed by Huffman coding, [Huf52], see Theorem 2.3.21.

In the case that $k > j + 1$, we have $N(C_{i,j,k}) = \frac{j-i}{2} + k - j + 1 \geq 2$. Hence, we may assume that for all $0 \leq i' \leq j' \leq k' \leq m - 1$ with $j' - i'$ even and $N(f_{i',j',k'}) < N(f_{i,j,k})$, a circuit $C_{i',j',k'}$ realizing $f_{i',j',k'}$ has already been computed. We may use these circuits while applying the recursion formulas (6.4), (6.5) and (6.6) with any valid λ in order to build a circuit for $f_{i,j,k}$. Note that split (6.4) requires a circuit for the function $f^*_{j+1,j+2\lambda+1,k}$ which can be obtained easily by dualizing the circuit $C_{j+1,j+2\lambda+1,k}$. The list $\mathcal{C}$ of candidate circuits in line 6 is never empty since in the case that $k > j + 1$, split (6.4) with $\lambda = 0$ is always a valid split.

Consequently, Algorithm 6.1 always computes a circuit $C_{0,m-1,m-1}$ realizing the And-Or path $f_{0,m-1,m-1}$ on inputs $t = (t_0, \dots, t_{m-1})$. Using the following observation, we will show that the delay of $C_{0,m-1,m-1}$ is always at least as good as the delay of the circuit computed by Algorithm 4.1.

Observation 6.1.2. For any $i, j, k \in \{0, \dots, m - 1\}$ with $i \leq j \leq k$ even, Algorithm 6.1 computes a realization $C_{i,j,k}$ with best possible delay among all circuits arising from the construction of optimum symmetric trees on input vectors of the form $t_i, t_{i+2}, \dots, t_{j-2}, t_j$ or $t_i, t_{i+2}, \dots, t_{j-2}, t_j, t_{j+1}$, the recursive application of (6.4) to (6.6) and the dualization of circuits.

Proposition 6.1.3. *Let Boolean input variables $t = (t_0, \dots, t_{m-1})$ with arrival times $a(t_0), \dots, a(t_{m-1}) \in \mathbb{N}$ be given. Consider the circuits C and C' computed by Algorithm 6.1 and Algorithm 4.1 for this instance, respectively. Then, we have*

$$\operatorname{delay}(C) \leq \operatorname{delay}\big(C'\big)\,.$$

Algorithm 6.1: Binary-circuit dynamic program for delay optimization of And-Or paths

Input: Boolean input variables $t = (t_0, \dots, t_{m-1})$ with arrival times $a(t_0), \dots, a(t_{m-1}) \in \mathbb{N}$.

Output: A circuit over Ω_{mon} computing $f_{0,0,m-1}$.

```
1 for $l \leftarrow 1$ to $m$ do
2     for $0 \leq i \leq j \leq k < m$ s.t. $j - i$ even and $N(f_{i,j,k}) = l$ do
3         if $k \in \{j, j+1\}$ then
4             $C_{i,j,k}$ := circuit for $f_{i,j,k}$ computed by Huffman coding [Huf52]
              (see Theorem 2.3.21)
5         else
6             $\mathcal{C}$ := list of candidate circuits for $f_{i,j,k}$ arising from applying split
              (6.4), (6.5) or (6.6) with any valid $\lambda$
7             $C_{i,j,k} := \operatorname{argmin}\{ d(C) : C \in \mathcal{C} \}$
8 return $C_{0,0,m-1}$
```

Proof. Assume that Algorithm 4.1 is applied for the realization of the And-Or path $g\big((t_0, \dots, t_{m-1})\big) = f_{0,0,m-1}$. Here, by Observation 4.1.18, the recursive call in line 19 can be avoided by instead using split (4.27). We show that with this modification, all recursive constructions of Algorithm 4.1 can be expressed by (possibly recursive applications of) the splits Equations (6.4) to (6.6):

- The symmetric split in line 10 is split (6.6) with $\lambda = \frac{j-i}{2}$.
- The split in line 14 is an alternating split (6.4) with a prefix of length 1.
- The split in line 29 is the alternating split as in Equation (6.4).
- The split (4.27) which is used to avoid the recursive call in line 19 is an alternating split with an even prefix as in Equation (6.5).

This shows that whenever Algorithm 4.1 (modified according to Observation 4.1.18) is applied recursively, the realized function is of the form $f_{i,j,k}$ for some $0 \leq i \leq j \leq k < m - 1$ with $j - i$ even.

Now we verify that each explicit construction in Algorithm 4.1 can also be found by Algorithm 6.1:

- The binary trees in line 4 of Algorithm 6.1 can be computed in line 4 of Algorithm 6.1.
- The realization $g(t) = t_0 \wedge (t_1 \vee t_2)$ computed in line 7 of Algorithm 6.1 can be obtained by applying Equation (6.4) with prefix length 1 to $g(t)$, and then by using the symmetric tree $t_1 \vee t_2$ which is dual to one of the symmetric trees constructed in line 4.
- The construction in line 26 is an alternating split (6.4) with an odd prefix of length 1.

Hence, all explicit and recursive constructions in Algorithm 4.1 can be performed by Algorithm 6.1. By Observation 6.1.2, this implies $\text{delay}(C) \leq \text{delay}(C')$. $\square$

The way Algorithm 6.1 is formulated, we construct a formula circuit for $f_{0,0,m-1}$. However, in practice, we try to avoid building the same sub-circuit twice and instead re-use the function computed by its output gate, see, e.g., the circuit in Figure 6.2(a). Our precise size improvement strategy is presented in Section 6.1.4, and in Section 6.2, we will see that in practice, our circuits seem to have a linear size. From a theoretical view, we can show that the size of the formula circuit constructed by Algorithm 6.1 is at most quadratic in the number of inputs.

Proposition 6.1.4. *Let Boolean input variables $t = (t_0, \dots, t_{m-1})$ with arrival times $a(t_0), \dots, a(t_{m-1}) \in \mathbb{N}$ be given. Consider $0 \leq i \leq j \leq k$ with $j - i$ even. Then, the circuit $C_{i,j,k}$ computed by Algorithm 6.1 has size at most $(k - j + 1)(k - i + 1) - 1$. In particular, the circuit $C_{0,0,m-1}$ for $f_{0,0,m-1}$ computed by Algorithm 6.1 has size at most $m^2 - 1$.*

Proof. The second statement is a special case of the first statement. We prove the first statement by induction on $N(f_{i,j,k})$.

If $k \in \{j, j+1\}$, then we construct $C_{i,j,k}$ as a binary tree with size at most $k - i$ in line 4. As $k \geq j$, this can be bounded from above by $(k - j + 1)(k - i + 1) - 1$, so the size bound is fulfilled.

This covers the case $N(f_{i,j,k}) \leq 1$, so assume now that $N(f_{i,j,k}) \geq 2$, where $C_{i,j,k}$ is constructed in lines 6 to 7. We consider three different cases based on the type of split that is used to construct $C_{i,j,k}$.

First assume that $C_{i,j,k} = C_{i,j,j+2\lambda} \wedge C^*_{j+1,j+2\lambda+1,k}$ for some $0 \leq \lambda \leq \frac{k-j-1}{2}$ as in Equation (6.4). Then, we have

$$\begin{aligned}
& \operatorname{size}(C_{i,j,k}) \\
= \; & \operatorname{size}\big(C_{i,j,j+2\lambda}\big) + \operatorname{size}\Big(C^*_{j+1,j+2\lambda+1,k}\Big) + 1 \\
\overset{\text{(IH)}}{\leq} \; & (j + 2\lambda - j + 1)(j + 2\lambda - i + 1) + (k - (j + 2\lambda + 1) + 1)(k - (j + 1) + 1) - 1 \\
= \; & (k - j + 1) \max\big\{ j + 2\lambda - i + 1, k - (j + 1) + 1 \big\} - 1 \\
\overset{\substack{j+2\lambda \leq k, \\ j \geq i}}{\leq} \; & (k - j + 1)(k - i + 1) - 1\,.
\end{aligned}$$

Now assume that $C_{i,j,k} = C_{i,j,j+2\lambda-1} \vee C_{i,j+2\lambda,k}$ for some $1 \leq \lambda \leq \frac{k-j}{2}$ as in Equation (6.5). Then, similar to the first case, we obtain

$$\begin{aligned}
& \operatorname{size}(C_{i,j,k}) \\
\leq \; & \operatorname{size}\big(C_{i,j,j+2\lambda-1}\big) + \operatorname{size}\big(C_{i,j+2\lambda,k}\big) + 1 \\
\overset{\text{(IH)}}{\leq} \; & (j + 2\lambda - 1 - j + 1)(j + 2\lambda - 1 - i + 1) + (k - (j + 2\lambda) + 1)(k - i + 1) - 1 \\
\overset{\substack{j+2\lambda \leq k, \\ j \geq i}}{\leq} \; & (k - j + 1) \max\{ j + 2\lambda - i, k - i + 1 \} - 1 \\
\overset{j+2\lambda \leq k}{=} \; & (k - j + 1)(k - i + 1) - 1\,.
\end{aligned}$$

Finally, in case of the split $C_{i,j,k} = C_{i,i+2\lambda-2,i+2\lambda-2} \wedge C_{i+2\lambda,j,k}$ with $1 \leq \lambda \leq \frac{j-i}{2}$ as in Equation (6.6), we have

$$\begin{aligned}
\text{size}(C_{i,j,k}) &= \text{size}\big(C_{i,i+2\lambda-2,i+2\lambda-2}\big) + \text{size}\big(C_{i+2\lambda,j,k}\big) + 1 \\
&\overset{\text{(IH)}}{\leq} 1 \cdot (i + 2\lambda - 2 - i + 1) + (k - j + 1)(k - (i + 2\lambda) + 1) - 1 \\
&= (2\lambda - 1) + (k - j + 1)(k - i - 2\lambda + 1) - 1 \\
&\overset{k \geq j}{<} (k - j + 1)(k - i) - 1\,.
\end{aligned}$$

This proves the induction step and hence the first statement. □

The following theorem summarizes all important properties of Algorithm 6.1.

Theorem 6.1.5. *Given Boolean input variables $t = (t_0, \ldots, t_{m-1})$ with arrival times $a(t_0), \ldots, a(t_{m-1}) \in \mathbb{N}$, Algorithm 6.1 computes a circuit C realizing the* AND-OR *path $g(t) = f_{0,0,m-1}$ with*

$$\text{delay}(C) \leq \log_2 W + \log_2 \log_2 m + \log_2 \log_2 \log_2 m + 4.3$$

and

$$\text{size}(C) \leq m^2 - 1$$

in running time $\mathcal{O}(m^4)$.

Proof. The size bound is proven in Proposition 6.1.4. We now prove the delay bound.

Let C' denote the circuit computed by Theorem 4.2.4 on the same instance. We show that $\text{delay}(C; a) \leq \text{delay}(C'; a)$, following the proof of Theorem 4.2.4.

For $m < 500$, C' is a standard AND-OR path circuit or the circuit computed by the algorithm by Held and Spirkl [HS17b] (for modified arrival times).

First assume that C' is the standard AND-OR path circuit. Note that the standard AND-OR path circuit for $f_{0,0,m-1}$ can be created by recursive application of Equation (6.4) with $\lambda = 0$ and finally constructing the symmetric circuit $f_{m-2,m-2,m-1}$. Hence, by Observation 6.1.2, we have $\text{delay}(C; a) \leq \text{delay}(C'; a)$.

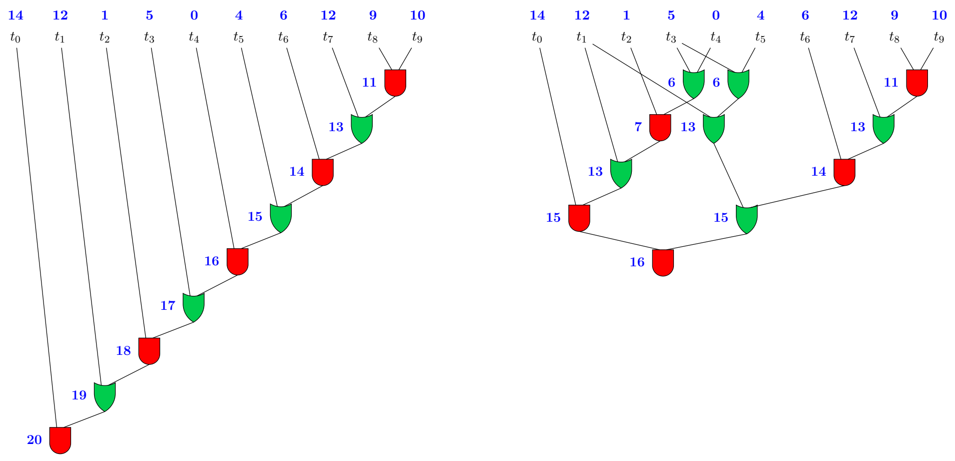
Now assume that C' is computed by Held and Spirkl [HS17b]. Recall from Section 2.6.4 that this circuit arises from recursive application of split (2.42) which is a special case of the alternating split (6.4). Note that Held and Spirkl [HS17b] construct special binary trees that follow the recursion of the alternating split, but as we always compute optimum binary trees in line 4, Observation 6.1.2 still implies $\text{delay}(C; a) \leq \text{delay}(C'; a)$.

For $m \geq 500$, the circuit C' is computed by Algorithm 4.1 (page 119) (on modified arrival times). In the proof of Proposition 6.1.3, we have seen that all initial and recursive circuit constructions used to construct C' can also be performed by Algorithm 6.1. Hence, by Observation 6.1.2, we have $\text{delay}(C; a) \leq \text{delay}(C'; a)$.

From $\text{delay}(C; a) \leq \text{delay}(C'; a)$ and Theorem 4.2.4, the delay bound follows.

In order to derive the running time bound, note that line 4 is executed $\mathcal{O}(m^2)$ times, while lines 6 and 7 are executed $\mathcal{O}(m^3)$ times. By Theorem 2.3.21, Huffman coding can be implemented in time $\mathcal{O}(m)$ after sorting and in time $\mathcal{O}(m \log_2 m)$ if sorting is needed. For a single execution of lines 6 and 7, the running time is in the order of $\mathcal{O}(m)$ as there are 3 types of splits and at most m choices for λ per split. Hence, the total running time is in $\mathcal{O}(m^2 \cdot m \log_2 m + m^3 \cdot m) = \mathcal{O}(m^4)$. □

In Section 6.2, we shall see that for most small instances, the solution computed by Algorithm 6.1 is much better than the solution computed by Theorem 4.2.4, the algorithm from Held and Spirkl [HS17b] or Rautenbach, Szegedy, and Werber [RSW06], see Figure 6.6 (page 179). However, the comparison with optimum delays in Figure 6.8 (page 182) shows that there is still much room for improvement. Thus, we will present a refined algorithm in Section 6.1.2. The following example shows that binary trees occurring in the middle of the circuit cannot be optimized by Algorithm 6.1; but it also explains the structure of circuits computed by Algorithm 6.1 on an example instance.


14 12 1 5 0 4 6 12 9 10
t_0 t_1 t_2 t_3 t_4 t_5 t_6 t_7 t_8 t_9
11
13
14
15
16
17
18
19
20
14 12 1 5 0 4 6 12 9 10
t_0 t_1 t_2 t_3 t_4 t_5 t_6 t_7 t_8 t_9
6
6
11
7
13
13
13
14
15
15
16

(a) The standard And-Or path circuit for $f_{0,0,9}$ with prescribed input arrival times and computed gate arrival times.

(b) A circuit for the instance on the left with delay 16 computed by Algorithm 6.1 as described in Example 6.1.6.

Figure 6.1: Applying Algorithm 6.1 (page 162) to compute the And-Or path $f_{0,0,9}$.

Example 6.1.6. Figure 6.1(b) depicts the solution computed by Algorithm 6.1 when run on the instance from Figure 6.1(a). The structure of the solution can be described as follows: At the output gate, we see that the alternating split with an odd prefix (see Equation (6.4)) has been applied with $\lambda = 2$:

$$f_{0,0,9} = f_{0,0,4} \wedge f^*_{1,5,9}$$

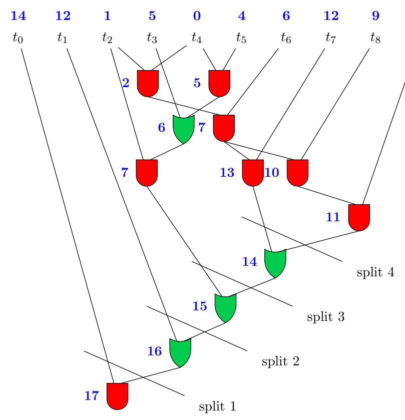
The sub-function $f_{0,0,4}$ is realized by the standard circuit, while $f^*_{1,5,9}$ is realized by the alternating split with an odd prefix (see Equation (6.4)) with $\lambda = 0$:

$$f^*_{1,5,9} = f^*_{1,5,5} \vee f_{6,6,9}$$

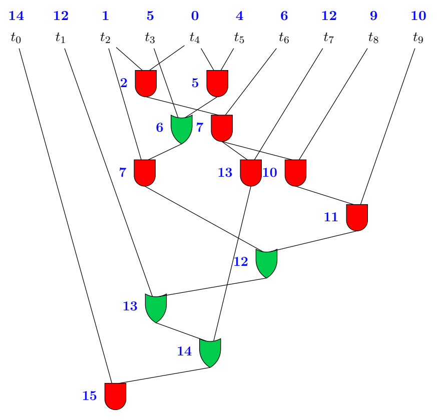
Figure 6.2(a) depicts a candidate solution contained in $\mathcal{C}$ for $f_{0,0,9}$ which is not delay-optimum and thus not output by the algorithm. To simplify explanations, we have marked important splits in the picture.

Splits 1 and 2: In these cases, the alternating split with an odd prefix (see Equation (6.4)) is used with $\lambda = 0$:

$$f_{0,0,9} = f_{0,0,0} \wedge f^*_{1,1,9}$$
$$f^*_{1,1,9} = f^*_{1,1,1,} \vee f_{2,2,9}$$



(a) Another circuit implementing the AND-OR path from Figure 6.1(a) which has been considered by Algorithm 6.1, but has delay 17.



(b) Circuit with optimum delay 15 arising from the circuit in Figure 6.2(a) by performing Huffman coding on the group of OR gates.

Figure 6.2: The circuit on the left-hand side is a candidate solution of Algorithm 6.1 (page 162) for $f_{0,0,9}$ obtained by the recursion formulas (6.4) to (6.6) as described in Example 6.1.6. The circuit on the right-hand side cannot be computed by Algorithm 6.1, but is delay-optimum for the instance in Figure 6.1(a) as the critical input t_0 traverses only 1 gate, which is best possible.

There is nothing to be done for the computation of $f_{0,0,0} = t_0$ or $f^*_{1,1,1,} = t_1$.

Split 3: Now, we apply the alternating split with an even prefix (see Equation (6.5)) with $\lambda = 2$:

$$f_{2,2,9} = f_{2,2,5} \vee f_{2,6,9}$$

The sub-function $f_{2,2,5}$ is realized by the standard circuit.

Split 4: Here, the alternating split with an even prefix (see Equation (6.5)) is applied with $\lambda = 1$:

$$f_{2,6,9} = f_{2,6,7} \vee f_{2,8,9}$$

As the two arising sub-functions are symmetric, they can be constructed using Huffman coding. Note that the circuit for $t_2 \wedge t_4 \wedge t_6$ is used in both sub-circuits.

In Figure 6.2(a), one can see very well where Algorithm 6.1 (page 162) lacks flexibility: Each drawn split line partitions the circuit into three parts: two sub-circuits, and a concatenation gate. The algorithm optimizes the two sub-circuits separately and does not cross these split lines during optimization. In this concrete example, re-arranging the OR concatenation gates as shown in Figure 6.2(b) is not possible for Algorithm 6.1. However, this would lead to a circuit with delay 15 which is by one better than the delay of the circuit in Figure 6.1(b) computed by Algorithm 6.1.

6.1.2 Undetermined-Circuit Dynamic Program

The drawback of Algorithm 6.1 (page 162) described in Example 6.1.6 leads us to refining Algorithm 6.1 with respect to symmetric trees whose gates belong to different sub-circuits. The resulting algorithm, which has been published previously by Brenner and Hermann [BH20], is presented in Algorithm 6.3. The key idea is to postpone the construction of symmetric trees until all their inputs are known. This motivates the following definition.

Definition 6.1.7. A circuit C with a single output is called an **undetermined circuit** if

- all its gates are And or Or gates and
- all gate vertices with the possible exception of $\operatorname{out}(C)$ have fan-in two.

We denote the gate type of $\operatorname{out}(C)$ by $\operatorname{gt}(C) \in \{\text{And}, \text{Or}\}$.

Note that any circuit over the basis $\Omega_{\text{mon}} = \{\text{And2}, \text{Or2}\}$ is an undetermined circuit. Another two undetermined circuits are shown in Figure 6.3.

Different to Algorithm 6.1 (page 162), we now allow undetermined circuits as implementations of the functions $f_{i,j,k}$ in intermediate solutions. Of course, the circuit we finally compute still must be a circuit over $\Omega_{\text{mon}} = \{\text{And2}, \text{Or2}\}$. We extend the definition of the weight of circuit inputs to undetermined circuits in order to compare different implementations realizing the same Boolean function.

Definition 6.1.8. Given an undetermined circuit C on Boolean inputs $t_0, \ldots, t_{m-1}$ with input arrival times $a(t_0), \ldots, a(t_{m-1}) \in \mathbb{N}$, we define the **weight** of C as

$$W(C) := \sum_{v \in \delta^{-1}(\operatorname{out}(C))} 2^{a(v)}\,.$$

Given the weight of an undetermined circuit, we can estimate the delay of a canonical logically equivalent circuit over Ω_{mon}.

Lemma 6.1.9. *Given an undetermined circuit C on inputs $t_0, \ldots, t_{m-1}$ with integral arrival times $a(t_0), \ldots, a(t_{m-1}) \in \mathbb{N}$, we can construct an equivalent circuit $\widetilde{C}$ over $\Omega_{\text{mon}} = \{\text{And2}, \text{Or2}\}$ with* $\operatorname{delay}\left(\widetilde{C}\right) = \lceil \log_2(W(C)) \rceil$.

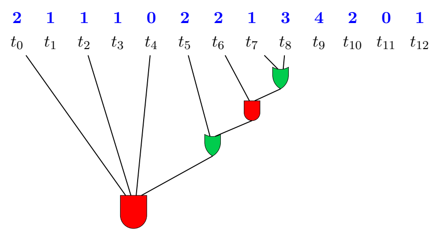
Proof. Initialize $\widetilde{C}$ with the circuit obtained from C by deleting $\operatorname{out}(C)$. Now, $\widetilde{C}$ is a circuit over Ω_{mon}, but has multiple outputs, say $v_0, \ldots, v_{k-1}$. Propagate the input arrival times through $\widetilde{C}$ to compute arrival times $a(v_0), \ldots, a(v_{k-1})$. Applying Huffman coding (see Theorem 2.3.21) with $v_0, \ldots, v_{k-1}$ as inputs and $a(v_0), \ldots, a(v_{k-1})$ as input arrival times yields a circuit H with delay $\lceil \log_2(W(C)) \rceil$. Adding all gates and edges of H to $\widetilde{C}$ yields the required circuit. $\square$

The rough idea of Algorithm 6.3 is again to compute a dynamic programming table which contains a circuit $C_{i,j,k}$ for $f_{i,j,k}$ for every $0 \leq i \leq j \leq k \leq m-1$ with $j-i$ even, but now, all intermediate solutions $C_{i,j,k}$ are undetermined circuits. Hence, we call Algorithm 6.3 the **undetermined-circuit dynamic program** in order to highlight the contrast to the binary-circuit dynamic program in Algorithm 6.1. Again, we apply the alternating splits (6.4) and (6.5) in order to recursively compute $C_{i,j,k}$, but the symmetric split (6.6) is used only with $\lambda = 1$, i.e.,

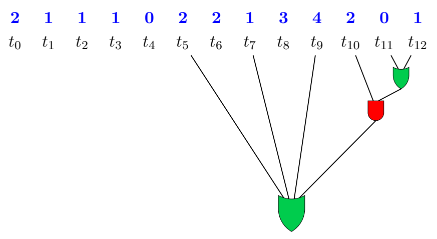
$$f_{i,j,k} = f_{i,i,i} \wedge f_{i+2,j,k}\,. \tag{6.7}$$

In the proof of Proposition 6.1.12, we shall see that the other symmetric splits are not needed anymore in Algorithm 6.3.

The three types of splits are extended to undetermined circuits as follows.

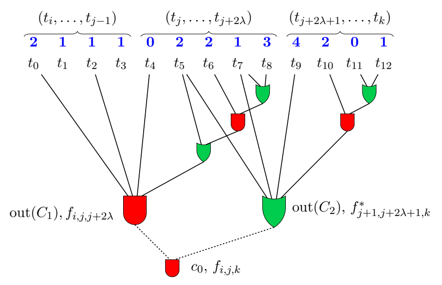

2 1 1 1 0 2 2 1 3 4 2 0 1
t_0 t_1 t_2 t_3 t_4 t_5 t_6 t_7 t_8 t_9 t_{10} t_{11} t_{12}

(a) An undetermined circuit for $f_{0,4,8}$ with weight $2^2 + 2^1 + 2^0 + 2^6 = 71$.

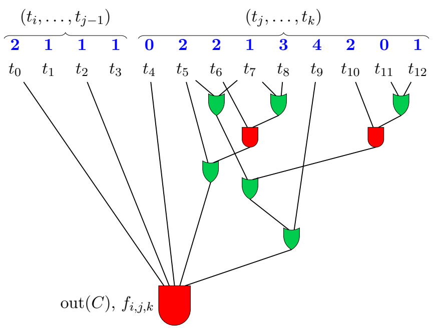

2 1 1 1 0 2 2 1 3 4 2 0 1
t_0 t_1 t_2 t_3 t_4 t_5 t_6 t_7 t_8 t_9 t_{10} t_{11} t_{12}

(b) An undetermined circuit for $f_{5,9,12}$ with weight $2^2 + 2^1 + 2^4 + 2^3 = 30$.

Figure 6.3: Two undetermined circuits and their weights.


$(t_i, \dots, t_{j-1})$
$(t_j, \dots, t_{j+2\lambda})$
$(t_{j+2\lambda+1}, \dots, t_k)$
2 1 1 1 0 2 2 1 3 4 2 0 1
t_0 t_1 t_2 t_3 t_4 t_5 t_6 t_7 t_8 t_9 t_{10} t_{11} t_{12}
out(C_1), $f_{i,j,j+2\lambda}$
out(C_2), $f^*_{j+1,j+2\lambda+1,k}$
c_0, $f_{i,j,k}$

(a) Combining the undetermined circuits from Figure 6.3 to a circuit for $f_{0,4,12}$ according to split (6.4) with $\lambda = 2$. This does not yield an undetermined circuit.



(b) Undetermined circuit C arising from applying Algorithm 6.2 to the circuits C_1 and C_2 from Figure 6.4(a). We have $W(C) = 2^2 + 2^1 + 2^0 + 2^6 + 2^5 = 103$.

Figure 6.4: Computing an undetermined circuit for $f_{i,j,k}$ with $i = 0$, $j = 4$ and $k = 12$ using the alternating split (6.4) with prefix-length $5 = 2\lambda + 1$ and Algorithm 6.2 (page 169).

Let us consider a circuit C for $f_{i,j,k}$ arising from the split

$$f_{i,j,k} = f_{i,j,j+2\lambda} \wedge f^*_{j+1,j+2\lambda+1,k}$$

with $\lambda \in \mathbb{N}$ and $0 \leq \lambda \leq \frac{k-j-1}{2}$ as in Equation (6.4). Now, the circuits $C_{i,j,j+2\lambda}$ for $f_{i,j,j+2\lambda}$ and $C^*_{j+1,j+2\lambda+1,k}$ for $f^*_{j+1,j+2\lambda+1,k}$ are both undetermined circuits. According to the split, using an AND gate, we can combine them to a circuit C' for $f_{i,j,k}$, but this will not necessarily be an undetermined circuit, see the example in Figure 6.4(a). In Figure 6.4(b), we can see how C' is turned into an undetermined circuit C for $f_{i,j,k}$ in this case.

The general procedure how to merge two undetermined circuits C_1 and C_2 is described in Algorithm 6.2: When $\text{gt}(C_i)$ coincides with the gate type $\circ$ of the concatenation gate, the inputs of $\text{out}(C_i)$ are simply connected to $\text{out}(C)$, otherwise, the symmetric tree at $\text{out}(C_i)$ may be completed using Lemma 6.1.9. This means that we do not decide for an implementation of the symmetric tree computing the logic function at $\text{out}(C_i)$ until we know that no other possible inputs of the symmetric tree will emerge in later steps of the algorithm.

In Algorithm 6.2, we see that the computed circuit C highly depends on whether

Algorithm 6.2: Merging undetermined circuits

```
Input: Two undetermined circuits $C_1$ and $C_2$ computing Boolean functions
       $h_1$ and $h_2$, respectively, depending on inputs with integer arrival
       times; a gate type $\circ \in \{\textsc{And}, \textsc{Or}\}$.
Output: An undetermined circuit $C$ computing $h_1 \circ h_2$.
1  Let $C$ be the union of the circuits $C_1$ and $C_2$.
2  Add a '$\circ$'-gate $v_0$ to $C$.
3  for $i \leftarrow 1$ to 2 do
4      Let $v_1, \ldots, v_k$ be the inputs of $\operatorname{out}(C_i)$.
5      if $\operatorname{gt}(C_i) = \circ$ then
6          Remove $\operatorname{out}(C_i)$ from $C$ and add edges $(v_1, v_0), \ldots, (v_k, v_0)$ to $C$.
7      else
8          Construct a circuit $\widetilde{C}_i$ over $\Omega_{\text{mon}}$ computing $h_i$ using Lemma 6.1.9.
9          Add all edges and gates from $\widetilde{C}_i$ to $C$.
10         Add an edge $\left(\operatorname{out}\left(\widetilde{C}_i\right), c_0\right)$ to $C$.
11     return $C$
```

$\operatorname{gt}(C_i) = \textsc{And}$ or $\operatorname{gt}(C_i) = \textsc{Or}$. As a consequence, our dynamic programming table now contains two undetermined circuits for every occurring $f_{i,j,k}$: circuits $A_{i,j,k}$ and $O_{i,j,k}$ which have minimum weight among all recursively constructed circuits with $\operatorname{gt}(A_{i,j,k}) = \textsc{And}$ and $\operatorname{gt}(O_{i,j,k}) = \textsc{Or}$, respectively. Here, we use the weight of the undetermined circuits to decide which circuit to store as by Lemma 6.1.9, two undetermined circuits C_1 and C_2 with $W(C_1) \le W(C_2)$ fulfill $\operatorname{delay}(C_1) \le \operatorname{delay}(C_2)$.

Apart from that, Algorithm 6.3 is very similar to Algorithm 6.1, but due to the use of undetermined circuits, we may omit some initial symmetric tree constructions.

The following lemma implies that Algorithm 6.3 correctly computes a circuit for $f_{0,0,m-1}$.

Lemma 6.1.10. *Consider the application of Algorithm 6.3 to Boolean input variables $t = (t_0, \ldots, t_{m-1})$ with arrival times $a(t_0), \ldots, a(t_{m-1}) \in \mathbb{N}$. Let $0 \le i \le j \le k < m$ with $j-i$ even be given. If $k-j \le 1$, we have $A_{i,j,k} = \bigwedge_{l \in \{i,i+2,\ldots,j-2,j\}} t_l$. Otherwise, both $A_{i,j,k}$ and $O_{i,j,k}$ exist.*

Proof. We prove the statement by induction on $N(f_{i,j,k}) \in \mathbb{N}$.

When $k - j = 0$, the circuit $A_{i,j,k} = \bigwedge_{l \in \{i,i+2,\ldots,j-2,j\}} t_l$ is computed in line 4.

For $k - j = 1$, a candidate circuit for $A_{i,j,k}$ can be obtained the following way: The realization $f_{i,j,k} = f_{i,j,j} \wedge f^*_{j+1,j+1,k}$ arises from the alternating split (6.4) with $\lambda = 0$ in line 6. By the first case, we have $A_{i,j,j} = \bigwedge_{l \in \{i,i+2,\ldots,j-2,j\}} t_l$ and $A_{j+1,j+1,k} = \bigwedge_{l=j+1} t_l = t_{j+1}$. After application of Algorithm 6.2, we obtain $A = \bigwedge_{l \in \{i,i+2,\ldots,j-2,j\}} t_l$ as a candidate circuit for $A_{i,j,k}$. As A has optimum weight among all possible realizations for $f_{i,j,k}$, we have $A_{i,j,k} = A$.

Now assume that $k-j \ge 2$, which implies $N(f_{i,j,k}) \ge 2$. By induction hypothesis, for all functions $f_{i',j',k'}$ such that $0 \le i' \le j' \le k' \le m-1$ with $j' - i'$ even and $N(f_{i',j',k'}) < N(f_{i,j,k})$, a realization $A_{i,j,k}$ is computed. Hence, a candidate realization for $A_{i,j,k}$ can be obtained via the alternating split Equation (6.4) with an odd prefix length $2\lambda + 1$ with $\lambda = 0$, i.e., $f_{i,j,k} = f_{i,j,j} \wedge f^*_{j+1,j+1,k}$. For $O_{i,j,k}$,

Algorithm 6.3: Undetermined-circuit dynamic program for delay optimization of AND-OR paths

```
Input: Boolean input variables $t = (t_0, \dots, t_{m-1})$ with arrival times
       $a(t_0), \dots, a(t_{m-1}) \in \mathbb{N}$.
Output: A circuit over $\Omega_{\text{mon}}$ computing $f_{0,0,m-1}$.
1  for $l \leftarrow 1$ to $m$ do
2      for $0 \leq i \leq j \leq k < m$ s.t. $j - i$ even and $N(f_{i,j,k}) = l$ do
3          if $k = j$ then
4              $A_{i,j,j} := \bigwedge_{l \in \{i, i+2, \dots, j-2, j\}} t_l$
5          $\mathcal{C}$ := list of candidate undetermined circuits for $f_{i,j,k}$ arising from
             applying split (6.4), (6.5) or (6.7) with any valid $\lambda$, followed by
             application of Algorithm 6.2.
6          $A_{i,j,k} = \operatorname{argmin}\{ W(C) : C \in \mathcal{C}$ with $\operatorname{gt}(C) = \text{AND} \}$
7          $O_{i,j,k} = \operatorname{argmin}\{ W(C) : C \in \mathcal{C}$ with $\operatorname{gt}(C) = \text{OR} \}$
8  if $m \leq 2$ then
9      $C := A_{0,0,m-1}$
10 else
11     $C := \operatorname{argmin}\{ W(A_{0,0,m-1}), W(O_{0,0,m-1}) \}$
12 Apply Lemma 6.1.9 to transform $C$ into a circuit $\tilde{C}$ over $\Omega_{\text{mon}}$ .
13 return $\tilde{C}$
```

a candidate circuit can be obtained via the alternating split Equation (6.5) with an even prefix length 2λ with $\lambda = 1$, i.e., $f_{i,j,k} = f_{i,j,j+1} \wedge f^*_{i,j+2,k}$. Hence, the list $\mathcal{C}$ in line 5 contains undetermined circuits both with AND and OR as output gate type, and $A_{i,j,k}$ and $O_{i,j,k}$ both exist. □

A similar statement as Observation 6.1.2 can be shown for Algorithm 6.3.

Observation 6.1.11. Let inputs variables $t = (t_0, \dots, t_{m-1})$ with arrival times $a(t_0), \dots, a(t_{m-1}) \in \mathbb{R}$ be given. We call a circuit C for $g(t)$ a **split circuit** if it arises from the construction of optimum symmetric trees on input vectors of the form $t_i, t_{i+2}, \dots, t_{j-2}, t_j$, the recursive application of (6.4), (6.5) and (6.7) followed by Algorithm 6.2 and the dualization of circuits. Moreover, we call a circuit C for $g(t)$ **split-optimum** if it is a split circuit that has optimum delay among all split circuits for $g(t)$. For integral arrival times, Algorithm 6.1 computes a split-optimum circuit for $g(t)$.

The following theorem states that regarding delay, the circuit computed by Algorithm 6.3 is always at least of good as the one computed by Algorithm 6.1.

Proposition 6.1.12. *Let Boolean input variables $t = (t_0, \dots, t_{m-1})$ with arrival times $a(t_0), \dots, a(t_{m-1}) \in \mathbb{N}$ be given. Consider the circuits C and $\widetilde{C}$ computed by Algorithm 6.1 and Algorithm 6.3 for this instance, respectively. Then, we have*

$$\operatorname{delay}(\widetilde{C}) \leq \operatorname{delay}(C).$$

Proof. For $0 \leq i \leq j \leq k < m$ with $j - i$ even, let $C_{i,j,k}$ be as in Algorithm 6.1 and $A_{i,j,k}$ and $O_{i,j,k}$ as in Algorithm 6.3. We will first prove the following claim.

Claim. For every $0 \leq i \leq j \leq k < m$ with $j - i$ even, we have

$$\operatorname{delay}(C_{i,j,k}) \geq \begin{cases} \lceil \log_2(W(A_{i,j,k})) \rceil & \text{if } \operatorname{gt}(C_{i,j,k}) = \text{And}, \\ \lceil \log_2(W(O_{i,j,k})) \rceil & \text{otherwise}. \end{cases}$$

Proof of claim: We prove the claim by induction on $N(f_{i,j,k})$.

If $k \in \{j, j+1\}$, Algorithm 6.1 constructs an optimum symmetric tree $C_{i,j,j}$ with $\operatorname{gt}(C_{i,j,k}) = \text{And}$ for $f_{i,j,k}$. In this case, by Lemma 6.1.10, we have $A_{i,j,j} := \bigwedge_{l \in \{i,i+2,\dots,j-2,j\}} t_l$. Thus, we have

$$\lceil \log_2(W(A_{i,j,j})) \rceil = \left\lceil \log_2 \left(\sum_{l \in \{i,i+2,\dots,j-2,j\}} 2^{a(t_l)} \right) \right\rceil = \operatorname{delay}(C_{i,j,j}).$$

Now assume that $k \geq j+2$. We have $N(f_{i,j,k}) = \frac{j-i}{2} + k - j + 1 \geq 2$. Hence, by induction hypothesis, we can assume that the statement holds for all $0 \leq i' \leq j' \leq k' \leq m-1$ with $j' - i'$ even and $N(f_{i',j',k'}) < N(f_{i,j,k})$,

The circuit $C_{i,j,k}$ is computed by a split of type (6.4), (6.5), or (6.3).

Case 1: Assume that the split is of type (6.4) or (6.5).

Let C_1 and C_2 denote the circuits used in the split by Algorithm 6.1. By Lemma 6.1.10, the table computed by Algorithm 6.3 contains circuits C_1' and C_2' where for each $r \in \{1,2\}$, C_i' is equivalent to C_i and $\operatorname{gt}(C_r) = \operatorname{gt}(C_r')$. By induction hypothesis, we have $\lceil \log_2(W(C_r')) \rceil \leq \operatorname{delay}(C_r)$ for each $r \in \{1,2\}$. Hence, the circuit C' arising from merging C_1' and C_2' by Algorithm 6.2 fulfills

$$W(C') \leq 2^{\lceil \log_2 W(C_1') \rceil} + 2^{\lceil \log_2 W(C_2') \rceil} \leq 2^{\operatorname{delay}(C_1)} + 2^{\operatorname{delay}(C_2)}.$$

Thus, assuming without loss of generality that $\operatorname{delay}(C_1) \leq \operatorname{delay}(C_2)$, this implies

$$\begin{aligned} \lceil \log_2(W(C')) \rceil &\leq \left\lceil \log_2\left(2^{\operatorname{delay}(C_1)} + 2^{\operatorname{delay}(C_2)}\right) \right\rceil \\ &\leq \left\lceil \log_2\left(2 \cdot 2^{\operatorname{delay}(C_2)}\right) \right\rceil \\ &\overset{\operatorname{delay}(C_2) \in \mathbb{N}}{=} \operatorname{delay}(C_2) + 1 \\ &= \max\{\operatorname{delay}(C_1), \operatorname{delay}(C_2)\} + 1 \\ &= \operatorname{delay}(C_{i,j,k}). \end{aligned} \tag{6.8}$$

Case 2: Assume that $C_{i,j,k}$ is computed via a symmetric split (6.3), i.e., $C_{i,j,k} = C_{i,i+2\lambda-2,i+2\lambda-2} \wedge C_{i+2\lambda,j,k}$ for some $1 \leq \lambda \leq \frac{j-i}{2}$.

An undetermined circuit $A_{i,j,k}'$ for $f_{i,j,k}$ can be obtained by recursive application of split of type (6.7), i.e.,

$$f_{i,j,k} = f_{i,i,i} \wedge \left(f_{i+2,i+2,i+2} \wedge \left(\dots \wedge \left(f_{i+2\lambda-2,i+2\lambda-2,i+2\lambda-2} \wedge f_{i+2\lambda,j,k} \right) \right) \right), \tag{6.9}$$

followed by Algorithm 6.2 after every split. As $A_{i,j,k}$ is a split-optimum circuit for $f_{i,j,k}$, we have $W(A_{i,j,k}) \leq W(A_{i,j,k}')$, and it suffices to show that

$$\left\lceil \log_2\left(W(A_{i,j,k}')\right) \right\rceil \leq \operatorname{delay}(C_{i,j,k}). \tag{6.10}$$

By Lemma 6.1.10, both $A_{i+2\lambda,j,k}$ and $O_{i+2\lambda,j,k}$ have been computed, and a trivial realization $A_{r,r,r} = t_r$ has been computed for all $r \in \{i, i+2, \dots, i+2\lambda-2\}$. Let $C'_{i+2\lambda,j,k}$ be the circuit among $A_{i+2\lambda,j,k}$ and $O_{i+2\lambda,j,k}$ with the same output gate type as $C_{i+2\lambda,j,k}$. Then, by induction hypothesis, we have

$$\left\lceil \log_2\Big(W(C'_{i+2\lambda,j,k})\Big) \right\rceil \leq \operatorname{delay}(C_{i+2\lambda,j,k})\,. \tag{6.11}$$

As all the outer gates in Equation (6.9) are And gates, we have

$$\begin{aligned} W(A'_{i,j,k}) \quad &\leq \quad \sum_{r \in \{i, i+2, i+2\lambda-2\}} W(t_r) + 2^{\left\lceil \log_2\left(W(C'_{i+2\lambda,j,k})\right)\right\rceil} \\ &\overset{\text{Alg. 6.1, l. 4}}{\leq} \quad 2^{\operatorname{delay}(C_{i,i+2\lambda-2,i+2\lambda-2})} + 2^{\left\lceil \log_2\left(W(C'_{i+2\lambda,j,k})\right)\right\rceil} \\ &\overset{(6.11)}{\leq} \quad 2^{\operatorname{delay}(C_{i,i+2\lambda-2,i+2\lambda-2})} + 2^{\operatorname{delay}\left((C_{i+2\lambda,j,k})\right)}\,. \end{aligned}$$

From this, we can show Equation (*6.10*) the same way as in Equation (6.8) of case 1.

This proves the induction step and hence the claim. □

In the case that $m \leq 2$, Algorithm 6.3 outputs the circuit $\widetilde{A}_{0,0,m-1}$ over Ω_{mon} arising from $A_{0,0,m-1}$ by application of Lemma 6.1.9. This is an optimum symmetric tree for $f_{i,j,k}$, so the statement holds.

When $m \geq 3$, Algorithm 6.3 outputs the circuit $\widetilde{C}$ over Ω_{mon} arising from the weight-optimum circuit among $A_{0,0,m-1}$ and $O_{0,0,m-1}$ by application of Lemma 6.1.9. From the claim, we hence deduce

$$\begin{aligned} \operatorname{delay}\Big(\widetilde{C}\Big) \quad &\overset{\text{Lem. 6.1.9}}{=} \quad \min\Big\{ \left\lceil \log_2(W(A_{0,0,m-1}))\right\rceil, \left\lceil \log_2(W(O_{0,0,m-1}))\right\rceil \Big\} \\ &\overset{\text{claim}}{\leq} \quad \operatorname{delay}(C_{0,0,m-1})\,. \end{aligned}$$ □

As in Proposition 6.1.4, we can show that if the circuits in Algorithm 6.3 are implemented as formula circuits, their size is at most quadratic. In practice, our circuits will have roughly linear size (see Section 6.2) as we heuristically optimize size as described in Section 6.1.4.

Proposition 6.1.13. *Let Boolean input variables $t = (t_0, \dots, t_{m-1})$ with arrival times $a(t_0), \dots, a(t_{m-1}) \in \mathbb{N}$ be given. Consider $0 \leq i \leq j \leq k$ with $j - i$ even. Let $A_{i,j,k}$ and $O_{i,j,k}$ (the latter only for $k \geq j+2$) be the undetermined circuits computed by Algorithm 6.3, and let $\widetilde{A}_{i,j,k}$ and $\widetilde{O}_{i,j,k}$ be the binary circuits arising from applying Lemma 6.1.9 to $A_{i,j,k}$ and $O_{i,j,k}$, respectively. Then, we have $\operatorname{size}(\widetilde{A}_{i,j,k})$, $\operatorname{size}(\widetilde{O}_{i,j,k}) \leq (k-j+1)(k-i+1)-1$. In particular, the circuit $C_{0,0,m-1}$ for $f_{0,0,m-1}$ computed by Algorithm 6.3 has size at most $m^2 - 1$.*

Proof. We prove the statement by induction on $N(f_{i,j,k})$. For $k \leq j+1$ (in particular for $N(f_{i,j,k}) \leq 1$), the circuit $A_{i,j,k}$ is a binary tree with size $k - i < (k-j+1)(k-i+1) - 1$ as $k = j$.

Thus, assume now that $N(f_{i,j,k}) \geq 2$ and $k \geq j+2$, where we construct $A_{i,j,k}$ and $O_{i,j,k}$ in lines 5 to 7. Without loss of generality, we only prove the statement for $A_{i,j,k}$. Assume that we use a split that builds $A_{i,j,k}$ from undetermined circuits C_1 and C_2. Note that the size of $\widetilde{A}_{i,j,k}$ is independent of the symmetric tree constructed for $\text{out}(A_{i,j,k})$. Thus, we can assume that $A_{i,j,k} = \widetilde{C}_1 \wedge \widetilde{C}_2$, where $\widetilde{C}_i$ arises from C_i by application of Lemma 6.1.9 for $i \in \{1,2\}$. Hence, we have $\text{size}(\widetilde{A}_{i,j,k}) \leq \text{size}(\widetilde{C}_1) + \text{size}(\widetilde{C}_2) + 1$. For $\widetilde{C}_1$ and $\widetilde{C}_2$, the induction hypothesis is fulfilled. From here, the proof of the induction step can be continued as in Proposition 6.1.4 as every split performed in Algorithm 6.3 is also performed in Algorithm 6.1. □

In the next theorem, we summarize the characteristics of Algorithm 6.3.

Theorem 6.1.14. *Given Boolean input variables $t = (t_0, \dots, t_{m-1})$ with arrival times $a(t_0), \dots, a(t_{m-1}) \in \mathbb{N}$, Algorithm 6.3 computes a split-optimum circuit $\widetilde{C}$ realizing the* And-Or *path $g(t) = f_{0,0,m-1}$ with delay at most*

$$\text{delay}(\widetilde{C}) \leq \log_2 W + \log_2 \log_2 m + \log_2 \log_2 \log_2 m + 4.3$$

and size at most

$$\text{size}(\widetilde{C}) \leq m^2 - 1$$

and can be implemented to run in time $\mathcal{O}(m^4)$.

Proof. Split-optimality of $\widetilde{C}$ holds by Observation 6.1.11, and the size bound is proven in Proposition 6.1.13. The delay guarantee follows from combining Proposition 6.1.12 and Theorem 6.1.5.

The running time guarantee is also implied by Theorem 6.1.5 since Algorithm 6.3 performs a strict subset of splits of Algorithm 6.1; only the running time of Algorithm 6.2 for the combination of sub-solutions by is additional (up to constant steps).

Note that during the course of Algorithm 6.3, we only need to know the weight and output gate type of an undetermined circuit, not its concrete structure. Hence, it suffices to actually construct symmetric trees in the final circuit $\widetilde{C}_{0,0,m-1}$ only. By postponing Huffman coding (Theorem 2.3.21), Algorithm 6.2 boils down to computing the output gate, summing up the weights of C_1 and C_2 and – eventually – rounding them up to the next power of 2. These tasks can be performed in constant time. Hence, lines 1 to 7 of Algorithm 6.3 can be implemented to run in time $\mathcal{O}(m^4)$ when no circuit is actually constructed.

For the construction of the final circuit $\widetilde{C}$, we now perform backtracking on the performed splits. For each split, we apply Algorithm 6.2, this time with application of Huffman coding [Huf52] (see Theorem 2.3.21), which takes time $\mathcal{O}(r \log_2 r)$ for each call on r inputs. As the size of $\widetilde{C}$ has been proven to be at most quadratic in m, we have $r \in \mathcal{O}(m^2)$ for each Huffman coding call. Hence, all Huffman coding calls together take time at most $\mathcal{O}(m^2 \log_2 m)$, which does not increase the overall running time.

We conclude that Algorithm 6.3 can be implemented to run in time $\mathcal{O}(m^4)$. □

Using Algorithm 6.3, we will in particular compute the optimum solution as depicted in Figure 6.2(b) for the instance from Example 6.1.6.

6.1.3 Extension to Fractional Arrival Times

In this section, we consider the And-Or Path Circuit Optimization Problem for instances $t = (t_0, \dots, t_{m-1})$ with fractional arrival times $a(t_0), \dots, a(t_{m-1}) \in \mathbb{R}$.

Our And-Or path optimization algorithm is dedicated to be used in the IBM design flow to optimize critical paths, see Chapter 7. We will see in Section 7.3.1 that the simple delay model regarded by our algorithm can be extended to incorporate physical attributes. But for this, we need fractional arrival times, where already the decimal places may represent, e.g., large distances or large gate delays. Hence, we do not round fractional arrival times to integers, but solve instances with fractional arrival times best possible, i.e., compute a split-optimum circuit (cf. Observation 6.1.11).

For instances with fractional arrival times, Algorithm 6.3 can also be applied to compute a split-optimum circuit, but with slight modifications which increase the running time by at least a factor of m. Later in this section, we show another approach which increases the running time only by a factor of $\log_2 m$.

First, we examine the SYMMETRIC FUNCTION DELAY OPTIMIZATION PROBLEM, which is a sub-problem we need to solve. Note that the weight of inputs is only defined for integral arrival times in Definition 2.3.16. The reason for this is that – as demonstrated in Example 2.3.24 – the delay of a symmetric circuit on inputs $x_0, \dots, x_{n-1}$ with arrival times $a(x_0), \dots, a(x_{n-1})$ cannot be determined from $\sum_{i=0}^{n-1} 2^{a(x_i)}$ only, different as in the case when all arrival times are integral. However, in Proposition 2.3.26, we have shown that Huffman coding [Huf52] can solve the SYMMETRIC FUNCTION DELAY OPTIMIZATION PROBLEM optimally also for fractional arrival times, in running time $\mathcal{O}(n \log_2 n)$. Furthermore, with

$$W_u(x; a) := \sum_{i=0}^{n-1} 2^{\lfloor a(x_i) \rfloor}, \tag{6.12}$$

by Observation 2.3.25, the optimum delay is in the interval

$$\Big[\Big\lceil \log_2 \big(W_u(x; a) \big) \Big\rceil, \Big\lceil \log_2 \big(W_u(x; a) \big) \Big\rceil + 1 \Big].$$

For Algorithm 6.3, this has the following consequences: In the sub-routine Algorithm 6.2, we cannot spare performing Huffman coding as we have done for integral arrival times in the proof of Theorem 6.1.14. Hence, Algorithm 6.2 requires at least linear running time. Furthermore, we need to store several circuits from the list $\mathcal{C}$ computed in line 5 to be sure to compute split-optimum circuits: For two candidate circuits $C_1 \neq C_2 \in \mathcal{C}$ with $\mathrm{gt}(C_1) = \mathrm{gt}(C_2)$, by Observation 2.3.25, we can only discard C_2 if $W_u(C_2; a) > 2W_u(C_1; a)$, where W_u from Equation (6.12) is extended to undetermined circuits as W is in Definition 6.1.8. Hence, we need to store more than one candidate per gate type for each $f_{i,j,k}$. Finally, in line 11, we need to choose a circuit with best delay among all candidate circuits for $f_{0,0,m-1}$.

The arising running time is hence at least in the order of $\Omega(m^5 \log_2 m)$, but in practice much higher due to the higher number of candidate circuits as demonstrated in Table 6.2.

With these modifications to Algorithm 6.3, we can ensure that the computed circuit is split-optimum also for non-integral arrival times. We call this algorithm the **explicit extension** of Algorithm 6.3 to fractional arrival times.

Proposition 6.1.15. *Let input variables $t = (t_0, \dots, t_{m-1})$ with fractional arrival times $a(t_0), \dots, a(t_{m-1}) \in \mathbb{R}$ be given. The explicit extension of Algorithm 6.3 to fractional arrival times computes a split-optimum circuit C for $g(t)$ in running time $\Omega(m^5)$.* □

As a faster and easier alternative, we can reduce the AND-OR PATH CIRCUIT OPTIMIZATION PROBLEM with fractional arrival times to its integral variant as in

Section 5.1. For this, given inputs $t = (t_0, \dots, t_{m-1})$, let S denote the set of split circuits for $g(t)$. Then, an algorithm A is S-optimum if and only if it computes a split-optimum circuit for $g(t)$. Hence, applying Proposition 5.1.3 or Theorem 5.1.5 with this set S and Algorithm 6.3, we can compute split-optimum And-Or path circuits for fractional arrival times:

Proposition 6.1.16. *Let input variables $t = (t_0, \dots, t_{m-1})$ with fractional arrival times $a(t_0), \dots, a(t_{m-1}) \in \mathbb{R}$ be given. The linear-search extension of Algorithm 6.3 computes a split-optimum* And-Or *path circuit for $g(t)$ in running time $\mathcal{O}(m^5)$.* □

Theorem 6.1.17. *Let input variables $t = (t_0, \dots, t_{m-1})$ with fractional arrival times $a(t_0), \dots, a(t_{m-1}) \in \mathbb{R}$ be given. The binary-search extension of Algorithm 6.3 computes a split-optimum circuit C for $g(t)$ in running time $\mathcal{O}(m^4 \log_2 m)$.* □

6.1.4 Heuristically Optimizing Circuit Size

Up to now, our And-Or path optimization algorithms presented in Algorithms 6.1 and 6.3 construct formula circuits. We will see in this section that the algorithms can be modified to compute a split-optimum circuit with size at most the optimum size of a split-optimum formula circuit. Moreover, we encourage the algorithms to save gates by re-using sub-circuits. Thus, we incorporate two steps for size improvement into both algorithms. We only describe these ideas for the case of Algorithm 6.3, for Algorithm 6.1, they can be applied similarly.

As a first heuristic, we avoid the construction of sub-circuits that realize the same Boolean function as, e.g., in Figures 6.2(a) and 6.2(b) (page 166). Here, we save 2 gates by using the sub-circuit computing the function $t_2 \wedge t_4 \wedge t_6$ twice.

Note that in Algorithm 6.3, all gates constructed are contained in binary trees which are realized using Huffman coding [Huf52], see Algorithm 2.1. Hence, during Huffman coding, we re-use sub-trees that have already been constructed. In order to enlarge the possibility that gates can be shared in multiple symmetric trees, we introduce a tie-breaker when choosing the vertices v and w to be merged in line line 4 of Algorithm 2.1. The tie-breaker is based on the set of inputs in the input cone of the respective vertices. E.g., in Figure 6.2(b) (page 166), we see that the symmetric trees $t_2 \wedge t_4 \wedge t_6$ and $t_2 \wedge t_4 \wedge t_6 \wedge t_8 \wedge t_9$ have a common prefix. Thus, for two vertices v_1 and v_2, in Huffman coding, we prefer merging v_1 over merging v_2 if $|\mathcal{I}_{v_1}| < |\mathcal{I}_{v_w}|$, or if $|\mathcal{I}_{v_1}| = |\mathcal{I}_{v_w}|$ and the smallest input index in $\mathcal{I}_{v_1}$ is smaller than the smallest input index in $\mathcal{I}_{v_w}$.

Our second measure for size improvement is to store more than one candidate undetermined circuit for the function $f_{i,j,k}$ for each possible output gate type in lines 6 and 7 of Algorithm 6.3. For each gate type, we keep a set of candidates with different delay and size properties which in particular contains a split-optimum solution. Then, during the circuit construction at the end of the algorithm, we can use smaller sub-circuits in delay-wise uncritical parts of the circuit.

We now describe in detail which candidate circuits for $f_{i,j,k}$ to store. The idea is to store a candidate circuit $C_{i,j,k}$ for $f_{i,j,k}$ if it is not dominated by any other candidate with respect to delay and size. As the circuit construction is performed at the very end of the algorithm, we do not know the accurate circuit sizes during the algorithm. Still, we can give an upper bound $\overline{s}(C_{i,j,k})$ on the size of $C_{i,j,k}$ by assuming that it is constructed as a formula circuit: For initial constructions, let $\overline{s}(C_{i,j,k}) = m + n - 1$, and for splits $C_{i,j,k} = C_1 \wedge C_2$, let $\overline{s}(C_{i,j,k}) = \overline{s}(C_1) + \overline{s}(C_2) + 1$. Now, we can define our dominance criterion.

Definition 6.1.18. Let Boolean input variables $t = (t_0, \dots, t_{m-1})$ with arrival times $a(t_0), \dots, a(t_{m-1}) \in \mathbb{N}$ and $0 \leq i \leq j \leq k < m$ with $j - i$ even be given. Consider two undetermined circuits C_1 and C_2 for $f_{i,j,k}$. We say that C_1 **dominates** C_2 if

$$\operatorname{gt}(C_1) = \operatorname{gt}(C_2) \text{ and } W(C_1) \leq W(C_2) \text{ and } \overline{s}(S_1) \leq \overline{s}(C_2)\,.$$

For each possible output gate type, we do not store only one candidate circuit $A_{i,j,k}$ or $O_{i,j,k}$, but a list of candidate undetermined circuits for $f_{i,j,k}$ that do not dominate each other regarding Definition 6.1.18. Hence, when applying a split for the computation of $f_{i,j,k}$, we need to combine any two non-dominated circuits for the respective sub-functions, and to store the result if it is not dominated by another candidate circuit.

This way, for each occurring weight, we keep the candidate undetermined circuit for $f_{i,j,k}$ with smallest size among all candidates with at most the same weight. In particular, when we finally choose an undetermined circuit for $f_{0,0,m-1}$, a weight-optimum circuit is contained among the candidates. As we actually optimize the delay of the corresponding binary circuit, we may choose an undetermined circuit for $f_{0,0,m-1}$ which has best size among all candidate undetermined circuits which lead to the optimum delay.

Using this size optimization strategy, the size of the circuit we compute is always at least as good as the best size of any split-optimum formula circuit for $f_{0,0,m-1}$.

As the size of the circuit $\widetilde{C}$ computed by Algorithm 6.3 is at most $m^2 - 1$ by Proposition 6.1.13, the number of different sizes that can occur for non-dominated circuits is bounded by this number. Hence, the number of non-dominated circuits for $f_{i,j,k}$ for fixed i, j, k is bounded by m^2. Thus, storing and merging all non-dominated candidates increases the total running time by at most a factor $\mathcal{O}(m^4)$. In Table 6.2, we will see that in practice, the running time increase is rather in the order of $\mathcal{O}(m^2)$.

Remark 6.1.19. Recall that in the explicit extension of Algorithm 6.3 to fractional arrival times described in Section 6.1.3, we use a different weight definition

$$W_u(C; a) = \left\lceil \log_2 \left(\sum_{v \in \delta^+(\operatorname{out}(C))} 2^{\lfloor a(v) \rfloor} \right) \right\rceil$$

from Equation (6.12), and that we already keep multiple candidate circuits for each $f_{i,j,k}$ in order to compute a split-optimum solution. Similarly, in Definition 6.1.18, we have to replace the weight criterion by $W_u(C_1) \leq 2W_u(C_2)$, which significantly increases the number of candidate circuits and hence the running time, see Table 6.2.

Both the binary and linear extension apply Algorithm 6.3 to integral arrival times, hence these are not affected by this remark.

6.2 Computational Results

In this section, we examine the quality of the circuits computed by our algorithms from Sections 6.1.1 and 6.1.2, compared with each other and previous And-Or path optimization algorithms, but also with our delay-optimum solutions from Chapter 5. All tests were executed on a machine with two Intel(R) Xeon(R) CPU E5-2699 v4 processors, using a single thread.

First, we examine small depth-optimization instances. In Table 6.1, for each $d \in \{0, \dots, 8\}$ and any stated And-Or path optimization algorithm, we show the

d	0	1	2	3	4	5	6	7	8
Algorithm 5.1	1	2	3	6	10	19	33	60	64*
Algorithm 6.1	1	2	3	6	10	19	33	60	109
Algorithm 6.3	1	2	3	6	10	19	33	60	109
[RSW06]	1	2	3	6	9	15	25	41	67
[HS17b]	1	2	3	5	8	14	24	40	66

Table 6.1: Maximum number m of inputs for which the respective method computes an And-Or path circuit with depth d. As Algorithm 5.1 works on 64-bit bitsets, the maximum instance it can solve has 64 inputs. Once this technicality is overcome, we expect to compute the optimum depth for up to 109 inputs within a few hours per instance.

maximum number of inputs m for which the respective algorithm computes an And-Or path circuit $g\big((t_0, \dots, t_{m-1})\big)$ with depth d.

As Algorithm 5.1 always computes depth-optimum solutions, from the table, we can conclude that both Algorithms 6.1 and 6.3 compute depth-optimum solutions for instances with up to 109 inputs. For 110 or more inputs, we need a depth of at least 9, and it is unknown whether this is best possible. A comparison with Table 5.7 would yield more instances where Algorithm 6.3 is optimum. In Section 6.3, we will examine the difference of Algorithms 6.1 and 6.3 to our exact algorithm and state in Conjecture 6.3.1 that we believe that both Algorithms 6.1 and 6.3 always compute optimum solutions regarding depth optimization.

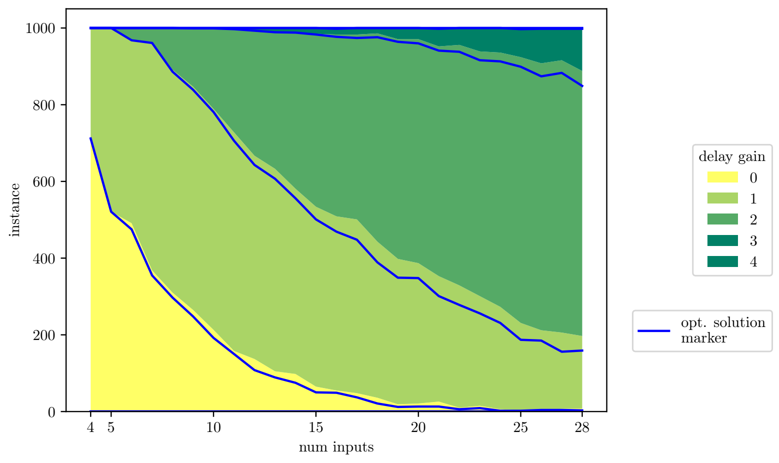
The algorithms by Rautenbach, Szegedy, and Werber [RSW06] and Held and Spirkl [HS17b] on many instances compute a solution which is by 1 worse than the optimum. In fact, for 110 inputs [HS17b] even needs depth 10 and is hence by 2 away from the optimum. Note that [RSW06] have an advantage over [HS17b] on our instances as they natively compute And-Or path of type $g\big((t_0, \dots, t_{m-1})\big)$ and not $g^*\big((t_0, \dots, t_{m-1})\big)$, so there are a few instances where they have a better depth.

Now, we examine instances with non-uniformal arrival times. Recall that by Proposition 5.2.6, for up to 3 inputs, the standard And-Or path circuit is an optimum solution to the And-Or Path Circuit Optimization Problem. Thus, instances with less than 4 inputs are not interesting for our comparisons. Furthermore, in our practical application in the IBM design flow (cf. Chapter 7), typical instances have up to 15 inputs, see Table 7.1.

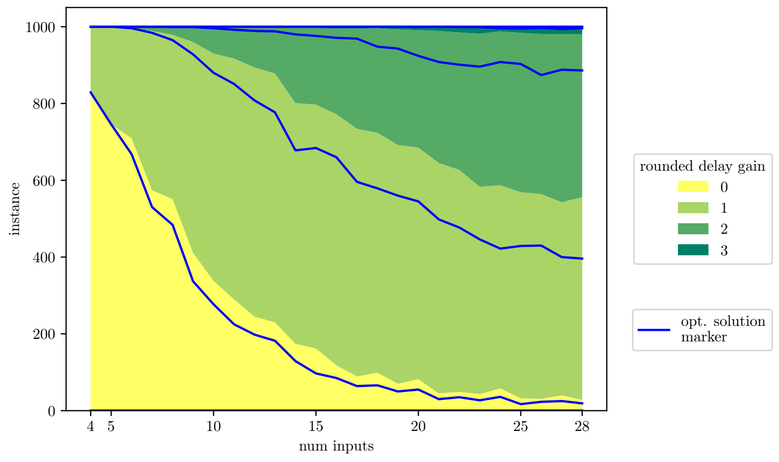
Hence, we computed two test sets TI and TF containing instances for the And-Or Path Circuit Optimization Problem with integral and fractional arrival times (with up to 4 digits), respectively. Each of the two test sets is of the following form: For each $n \in \{4, 28\}$, there are 1000 instances with n inputs with arrival times chosen uniformly from the interval $[0, n]$. Results look very similar when the arrival time patterns are more balanced, e.g., when we restrict the interval our random arrival times are chosen from to a range of $[0, \log_2 n + 2]$.

For each of the instances I in TI and TF, we computed

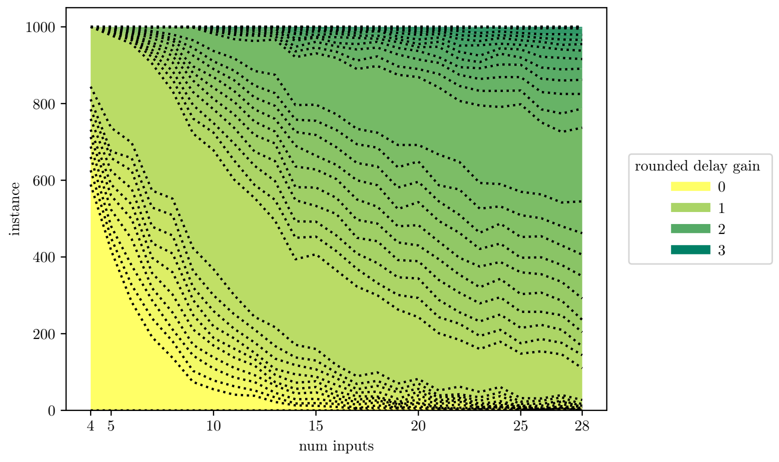
- the optimum delay $\text{opt}(I)$ computed by Algorithm 5.1,
- the circuit $C_B(I)$ (binary-circuit dynamic program) resulting from Algorithm 6.1 (page 162), with heuristic size optimization as in Section 6.1.4,



(a) Delay gains on test set TI.

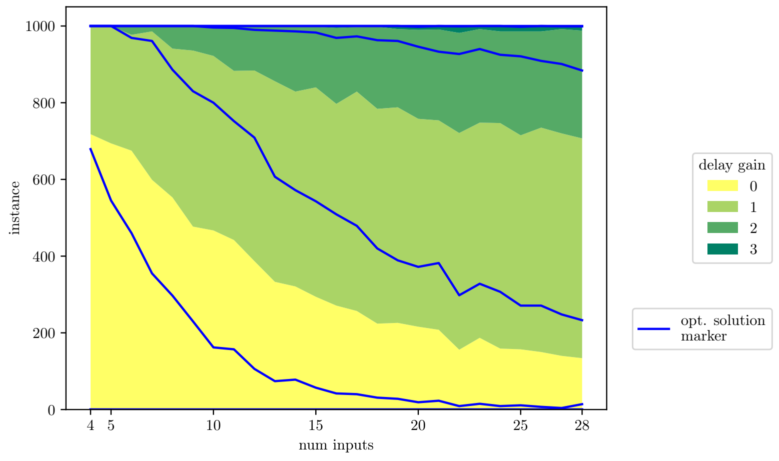


(b) Delay gains on test set TF, rounded down to integers.

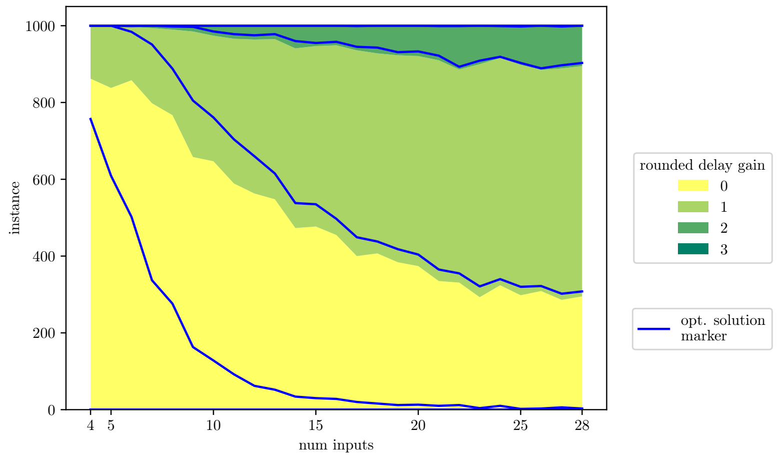


(c) Delay gains on test set TF, rounded down to multiples of 0.1.

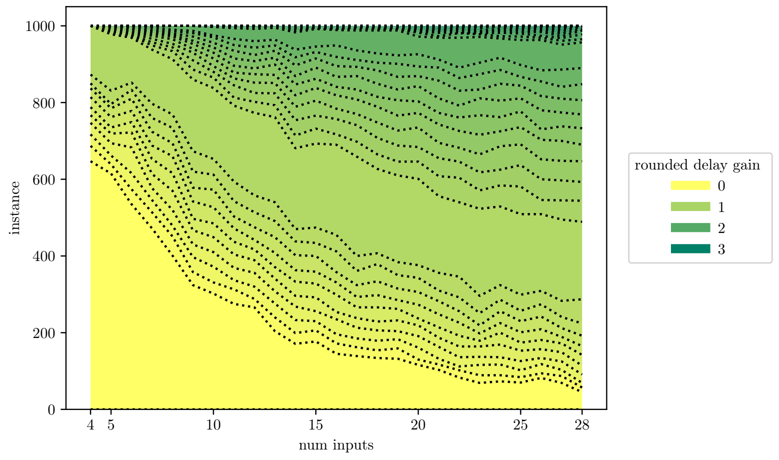
Figure 6.5: Delay gains of the circuits $C_U(I)$ computed by Algorithm 6.3 (page 170) over $C_P(I)$, the circuit with best delay among those computed by Held and Spirkl [HS17b] and Rautenbach, Szegedy, and Werber [RSW06], and the optimum delays $\operatorname{opt}(I)$ computed by Algorithm 5.1.



(a) Delay gains on test set TI.



(b) Delay gains on test set TF, rounded down to integers.



(c) Delay gains on test set TF, rounded down to multiples of 0.1.

Figure 6.6: Delay gain of the circuits $C_B(I)$ computed by Algorithm 6.1 (page 162) over $C_P(I)$, the circuit with best delay among those computed by Held and Spirkl [HS17b] and Rautenbach, Szegedy, and Werber [RSW06], and the optimum delays $\mathrm{opt}(I)$ computed by Algorithm 5.1.

- the circuit $C_U(I)$ (undetermined-circuit dynamic program) resulting from Algorithm 6.3 (page 170) with heuristic size optimization as in Section 6.1.4, where for fractional arrival times, we use the binary-search extension from Theorem 6.1.17,
- the circuit $C_P(I)$ (previous work) with best delay among those constructed by Held and Spirkl [HS17b] and Rautenbach, Szegedy, and Werber [RSW06].

Note that for these instance sizes, our algorithm from Theorem 4.2.4 will compute the same solution as [HS17b], so its delay will not be better than the delay of $C_P(I)$.

In Figure 6.5, we compare the circuits $C_B(I)$, $C_U(I)$, $C_P(I)$ and the optimum solution $\operatorname{opt}(I)$ for each instance I in test set TI and TF: Each coordinate in any of these plots corresponds to one instance, and instances are grouped by their numbers of inputs in vertical lines. The color of an instance I is chosen based on the (in case of TF rounded) delay gain of $C_U(I)$ over $C_P(I)$. More precisely, for the case of integral arrival times (see Figure 6.5(a)), instances are colored by $\operatorname{delay}(C_P(I)) - \operatorname{delay}(C_U(I))$; and for the case of fractional arrival times, they are colored by $\Big\lfloor \alpha\big(\operatorname{delay}(C_P(I)) - \operatorname{delay}(C_U(I))\big) \Big\rfloor$, where $\alpha = 1$ in Figure 6.5(b) and $\alpha = 10$ in Figure 6.5(c). In Figure 6.5(c), dotted lines separate the instance groups with different colors. The blue lines in Figures 6.5(a) and 6.5(b) indicate the portion of instances I for which $\operatorname{delay}(C_U(I)) = \operatorname{opt}(I)$: For all instances I in a colored part which are drawn below the blue line, the circuits $C_U(I)$ are delay-optimum, and for those above the blue line, they are not.

Recall from Sections 6.1.2 and 6.1.3 that Algorithm 6.3 and all its extensions to fractional arrival times compute split-optimum AND-OR path circuits as defined in Observation 6.1.11. In Section 2.6.2, we observed that all recursion options of [HS17b] and [RSW06] can be described using the splits used in Observation 6.1.11, so all delay gains are positive values. In general, the delay gain increases with increasing number of inputs.

As shown in the legend, the delay gain varies between 0 (yellow) and 4 (dark green). For testbed TI, there are 5 instances with a delay gain of 4, with numbers of inputs between 20 and 28. We see in Figure 6.5(a) that on a large fraction of the instances I in TI, the delay of $C_U(I)$ is better than the delay of $C_P(I)$, and on instances with at least 18 inputs, even better by 2 or more.

Figure 6.5(c) shows that on almost every instance in TF, $C_U(I)$ has better delay than $C_P(I)$, and for instances with at least 18 inputs, the delay of $C_U(I)$ is better than the delay of $C_P(I)$ by at least 1.4. However, even for 28 inputs, the fraction of the instances of TF with delay gain 2 or more is only about 3/5, and there is no instance with a delay gain of at least 4.

In Figures 6.5(a) and 6.5(b), we see that the delay of the circuits $C_U(I)$ – in contrast to the circuits $C_P(I)$ – is very often optimum. For test set TF, the delay difference of our algorithm looks much worse than for test set TF, but in Figure 6.8(b), we will see that most of the non-optimum circuits $C_U(I)$ only deviate from the optimum by at most 0.4. In fact, the average delay difference of the circuit $C_U(I)$ to $\operatorname{opt}(I)$ is roughly 0.04 on both testbeds.

Figure 6.6 compares the circuits $C_P(I)$ with the circuits $C_B(I)$ computed by Algorithm 6.1. We see that delay improvements of $C_B(I)$ over $C_P(I)$ are still high, but much lower than for $C_U(I)$. In particular, for the test set TF, there are barely any instances where $C_B(I)$ has optimum delay and $C_P(I)$ does not, and the amount of instances with a delay gain of ≥ 2 is very low.

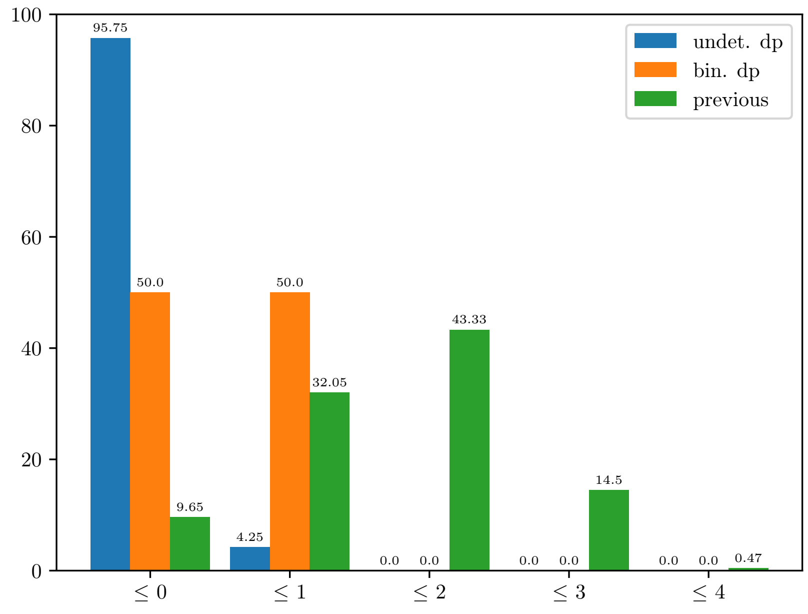


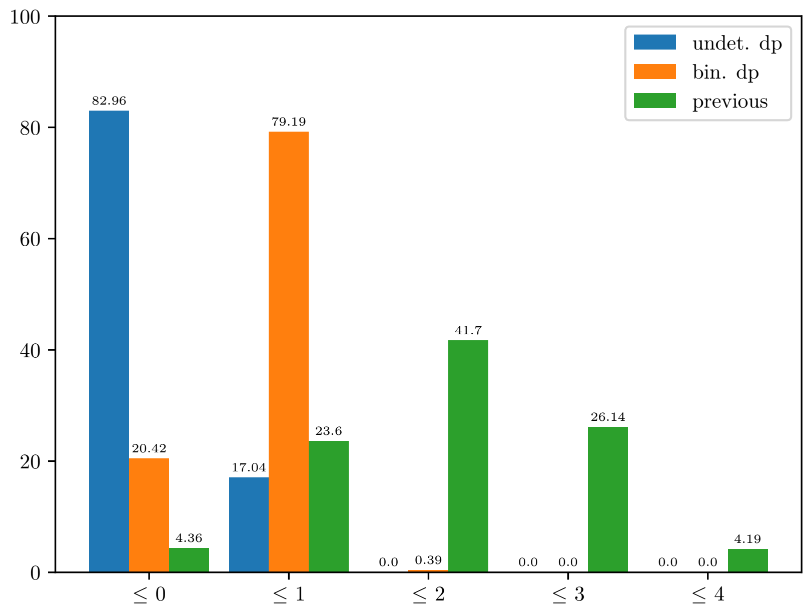
Figure 6.7: For any $d \in \{0, \dots, 5\}$, we show the portion of instances I in test set TI for which the delay difference of the circuit computed by the respective algorithm to the optimum solution $\mathrm{opt}(I)$ computed by Algorithm 5.1 is d. We compare the solution $C_U(I)$ computed by Algorithm 6.3 (page 170), the solution $C_B(I)$ computed by Algorithm 6.1 (page 162), and the circuit $C_P(I)$ with the best delay among those of [HS17b] and [RSW06].

Now, we analyze the delay differences of $C_U(I)$, $C_B(I)$ and $C_P(I)$ to the optimum solution in more detail. For the testbed TI, the delay differences are depicted in Figure 6.7. For each value d on the x-axis, we plot the percentage of the instances from testbed TI for which the respective circuits have delay difference exactly d to $\mathrm{opt}(I)$.

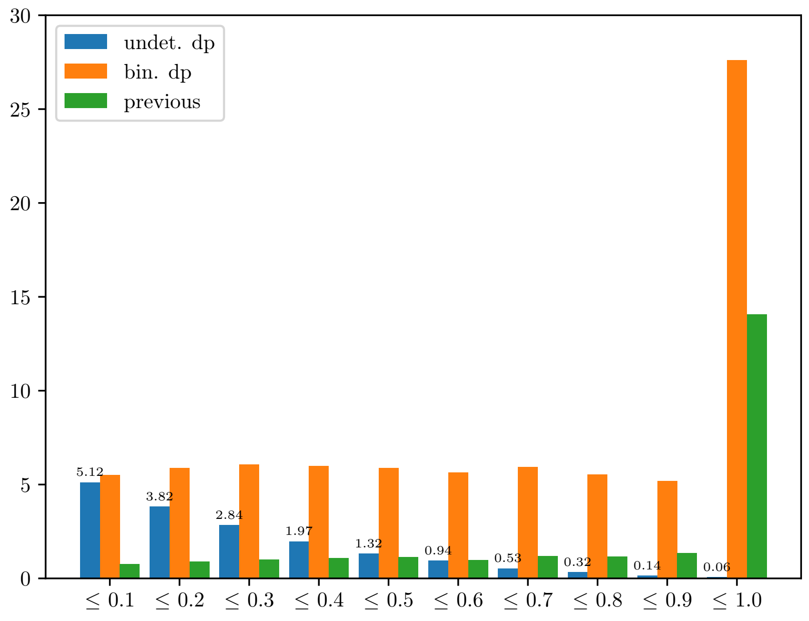
We see that on 95.75% of the integral instances, the delay of $C_U(I)$ is optimum, and that its delay difference to the optimum is always at most 1. The delay of $C_B(I)$ is optimum only for 50.00% of the instances, but also never away from the optimum by more than 1. In contrast, $C_P(I)$ differs from the optimum by up to 4, and on 43.33% of the instances by 2.

For the test set TF, we show two plots in Figure 6.8. Both show the rounded delay differences of the respective circuits to the optimum solution. In Figure 6.8(a), we show all instances and round delay differences down to integers. The comparison of the three algorithms looks similar as for the test set TI, but all algorithms seem to perform worse than on test set TI. However, there are many instances with a fractional delay difference which is rounded down in the plots.

Hence, in Figure 6.8(b), for every algorithm, we restrict ourselves to the part of Figure 6.8(a) with delay difference at most 1 and plot the delay differences rounded down to multiples of 0.1. Again, the y-axis displays the percentage of all instances which are solved with the rounded delay gain as on the x-axis. Now, we see that for $C_U(I)$, actually only 0.06 percent of the instances have a delay difference of 1 to the optimum, and that on more than 96 % of the instances, the delay difference to the



(a) Delay differences to the optimum solutions rounded up to integers for instances I in test set TF. All instances with delay difference 0 are solved delay-optimally.



(b) More detailed extract of Figure 6.8(a) in the interval (0, 1]. Here, delay differences are rounded up to multiples of 0.1.

Figure 6.8: For any $d \in \{0, \ldots, 5\}$, we show the portion of instances I in test set TF for which the rounded delay difference of the circuit computed by the respective algorithm to the optimum solution $\text{opt}(I)$ computed by Algorithm 5.1 is d. We compare the solution $C_U(I)$ computed by Algorithm 6.3 (page 170), the solution $C_B(I)$ computed by Algorithm 6.1 (page 162), and the circuit $C_P(I)$ with the best delay among those of [HS17b] and [RSW06].

optimum is at most 0.4.

On average, the delay difference of $C_U(I)$ to the optimum is about 0.04 both on testbeds TI and TF, while it is roughly 1.64 for $C_P(I)$ on TI and roughly 1.68 for $C_P(I)$ on TF.

We have verified that for all instances I in testbeds TI and TF with up to 5 inputs, our circuits $C_U(I)$ are always delay-optimum. However, already for 6 inputs, using our optimum algorithm, we discovered that there is an instance where $C_U(I)$ does not have optimum delay, see Figure 6.12 (page 189).

We shall now analyze the sizes and fanouts of our circuits $C_U(I)$ computed by Algorithm 6.3 (page 170). As for fractional arrival times, we use the binary-search extension of Algorithm 6.3, each solution computed is obtained from an instance with integral arrival times by modifying only arrival times. Hence, we only analyze sizes and fanouts of $C_U(I)$ for I in the testbed TI.

In Figure 6.9, we compare the sizes of our circuits $C_U(I)$ with those of $C_P(I)$ for all I in testbed TI. Each blue circle represents a set of instances (the number of instances being linear in the circle's area) with delay gain $\text{delay}(C_P(I)) - \text{delay}(C_U(I))$ as on the x-axis and size increase $\text{size}(C_U(I))/\,\text{size}(C_P(I))$ as on the y-axis. For each occurring delay gain, the black dot marks the average size increase for all instances with this delay gain.

In these experiments, during circuit construction, we always re-use existing gates in Huffman coding as described in Section 6.1.4. However, we will see in Table 6.2 that the other size improvement strategy described in Section 6.1.4 – storing all non-dominated candidates with respect to weight and size – is very time-consuming, so we also show circuit sizes when disabling this feature. This way, we still compute a split-optimum solution, but with worse circuit size.

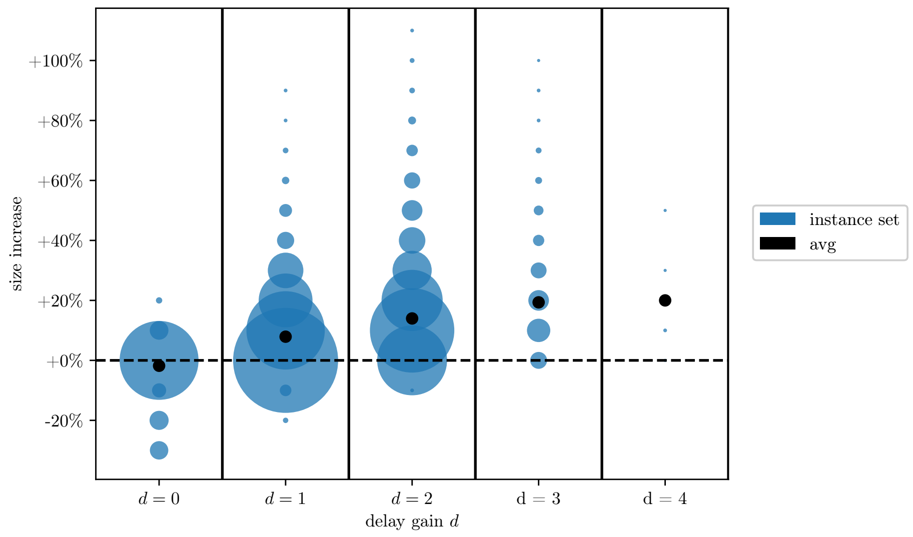
In Figure 6.9(a), our circuits $C_U(I)$ are computed with the heuristic size optimization strategy from Section 6.1.4. The number of gates used in our circuits $C_U(I)$ is typically higher than in the circuits $C_P(I)$, but mostly, the size increase is in range of 20%. Note that for instances with delay gain 0, the sizes of our circuits $C_U(I)$ are comparable with the sizes of the circuits $C_P(I)$, and on average, we even compute smaller circuits. The average size increase rises with increasing delay gain.

If we do not store all non-dominated candidates with respect to weight and size as in Section 6.1.4, we obtain the size increases as shown in Figure 6.9(b). Both the average and maximum size increase are much higher than in Figure 6.9(a), so this size improvement technique is very effective.

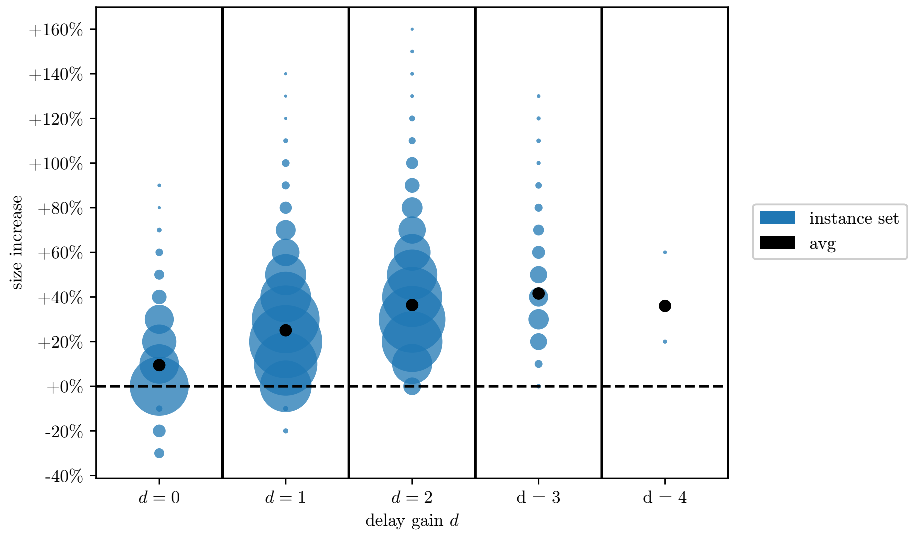
In Figure 6.10, we plot the fanout distribution in the circuits $C_U(I)$ for the instances I from test set TI. For each number of inputs on the x-axis, we gather all vertices with fanout at least 1 for all instances I with this number of inputs. Each vertex is colored by its fanout. In Figure 6.10(a), we plot all vertices with fanout at least 1. As here, vertices with fanout at least 4 can barely be seen, in Figure 6.10(b), we only plot those vertices.

First note that Figure 6.10(a) indicates that the total number of inputs and vertices (without outputs) grows roughly linearly with the number of inputs. Hence, in practice, the average size of our circuits $C_U(I)$ computed by Algorithm 6.3 (page 170) seems to be linear in the number of inputs, more precisely, in a range of about $1.5n$. For reference, for the circuits computed by Held and Spirkl [HS17b], the number of vertices is $1.2n$ on average.

Now, consider the fanouts. In Figure 6.10(a), we see that the majority of the vertices has fanout 1. Also considering Figure 6.10(b), the number of vertices



(a) Comparison of the sizes of the circuits $C_P(I)$ with the circuits $C_U(I)$ computed by Algorithm 6.3 (page 170) with the size heuristic size optimization from Section 6.1.4.



(b) Comparison of the sizes of the circuits $C_P(I)$ with the circuits $C_U(I)$ computed by Algorithm 6.3 (page 170) without storing all non-dominated candidates as in Section 6.1.4.

Figure 6.9: Comparison of the size (number of gates) of the circuit $C_P(I)$, the circuit with best delay among those computed by the algorithms from Held and Spirkl [HS17b] and Rautenbach, Szegedy, and Werber [RSW06], with the circuits $C_U(I)$ computed by Algorithm 6.3 (page 170) with or without storing all non-dominated candidate circuits regarding weight and size as in Section 6.1.4 for instances I in testbed TI.

with fanout k decreases rapidly with increasing k. As expected, the amount of vertices with high fanouts increases with increasing number of inputs. The maximum occurring fanout is 9, and less than 0.8% of all vertices have fanout at least 4.

This indicates that after buffering, our circuits will still have good delays. In Section 7.1, we will see that in our application in chip design, a fanout of mo re than 2 or 3 at a vertex v is often not desired in practice. Instead, often, a so-called buffer tree will be inserted after v in order to ensure that all gates have fanout at most 2. The circuits computed by Held and Spirkl [HS17b] and Rautenbach, Szegedy, and Werber [RSW06] in fact have a maximum fanout of 2.

Thus, in another experiment, we evaluated the delay of our circuits $C_U(I)$ after a simple buffering: For each vertex v with $k >= 3$ successors, we add a delay penalty of $\lceil \log_2 k \rceil - 1$. This assumes that a symmetric buffer tree is built; in practice, required arrival times at the successors would be respected in order to avoid long buffer chains on delay-critical parts of the circuit. Figure 6.11 depicts the delay gain of our circuits $C_U(I)$ over $C_P(I)$ with this delay model. As in $C_P(I)$, the maximum fanout is 2 anyway, the delay remains the same here; in our circuits, the delay is sometimes worse than with our usual delay model. But often, vertices with higher fanout are not critical, hence buffering does not influence the total delay.

In Figure 6.11, we see that on a tiny fraction of the instances I of testbed TI, the buffered delay of $C_P(I)$ is better than for our circuits $C_U(I)$, but only by 1. Still, for both test sets, on the majority of the instances, our circuits $C_U(I)$ have a better buffered delay than $C_P(I)$, and the overall distribution of the delay gains is not much worse than for the original delay model as in Figure 6.5(a).

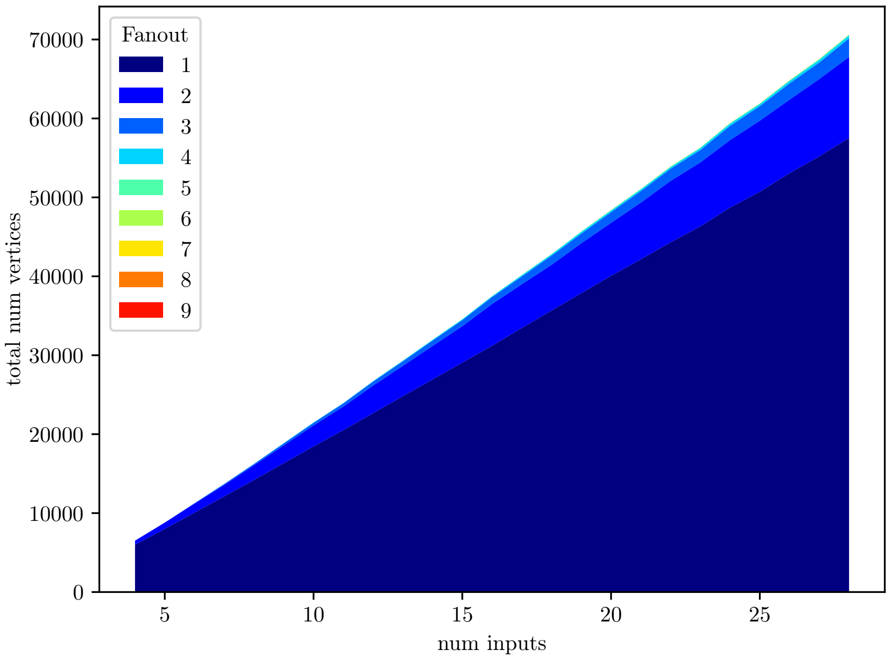
As our buffering penalty is pessimistic, we assume that the impact of buffering is much less in practice. Furthermore, one could incorporate buffering in Algorithm 6.3: Although it is an open question whether it is possible to optimize a buffered delay in Algorithm 6.3, it would not be hard to incorporate fanout in the dominance criterion from Definition 6.1.18 which is already used for size optimization.

Finally, in Table 6.2, we show a running time comparison of different versions of Algorithm 6.3. Here, both for integral and fractional arrival times, for each number $n \in \{10, 20, \ldots, 100\}$, we created 10 instances with random arrival times on n inputs in the interval $[0, n]$. These are precisely the testbeds used for the running time analysis of our exact algorithm from Chapter 5 in Table 5.3.

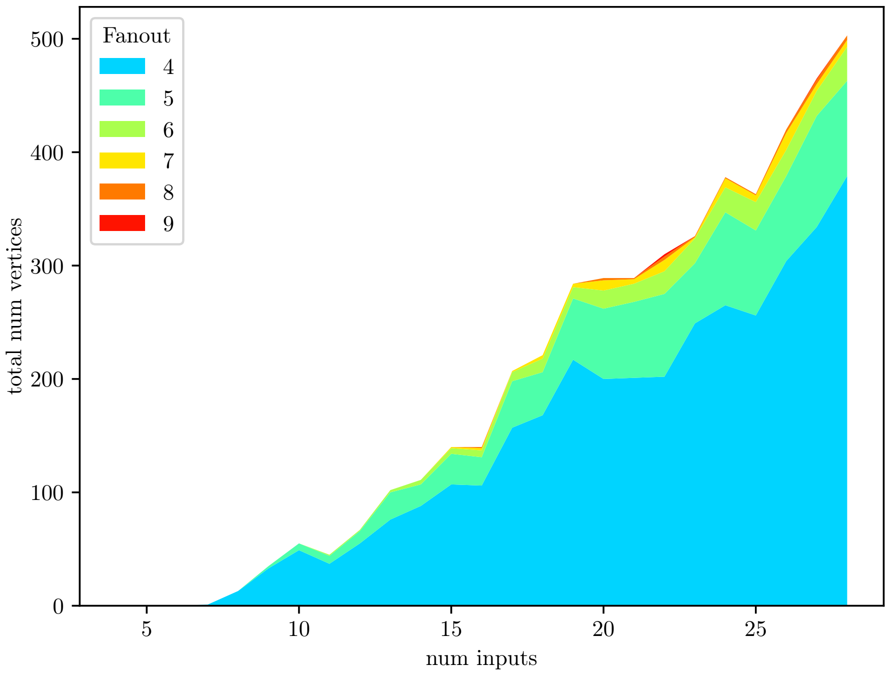
On each of the instances with integral arrival times, we ran Algorithm 6.3 with and without storing all non-dominated candidates regarding weight and size as in Section 6.1.4. On each instance with fractional arrival times, we ran the three extensions of Algorithm 6.3 to fractional arrival times from Section 6.1.3 with size improvement enabled: the binary-search extension from Theorem 6.1.17, the linear-search extension from Proposition 6.1.16, and the explicit extension from Proposition 6.1.15. In Table 6.2, in the row with $\#inputs = n$, we show the average running time of the respective algorithm on the instances with n inputs. We do not show the running time of the algorithm by Held and Spirkl [HS17b] as it is less than 0.001 seconds on any instance.

The value λ shown in the last row is a hint on the asymptotic running time of the respective algorithm A: Assuming that the running time of algorithm A is bounded by some polynomial π, and on this instance set, the running time is already dominated by π's variable with highest degree, the running time of algorithm A on n inputs will by roughly in the order of n^λ.

On instances with integral arrival times, we see that the average running time



(a) Fanout distribution of $C_U(I)$ for all instances I in test set TI.



(b) Figure 6.10(a) restricted to the vertices with fanout at least 4.

Figure 6.10: Fanout distribution of the circuits $C_U(I)$ computed by Algorithm 6.3 (page 170) for all instances of test set TI.

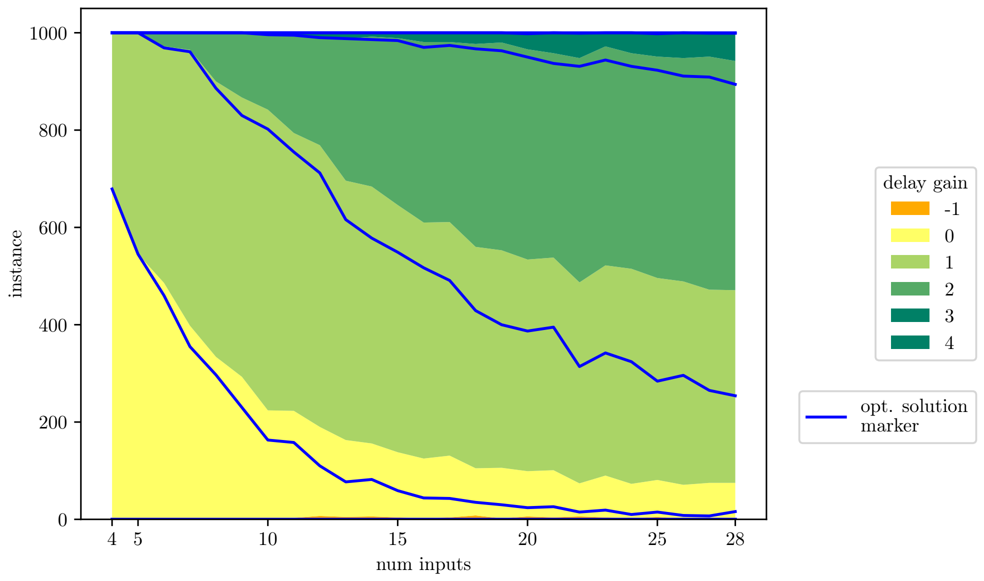


Figure 6.11: Comparison of the buffered delay of the circuit $C_U(I)$ computed by Algorithm 6.3 (page 170) with $C_P(I)$, the circuit with best delay among those computed by Held and Spirkl [HS17b] and Rautenbach, Szegedy, and Werber [RSW06], and the optimum delay $\text{opt}(I)$ computed by Algorithm 5.1 on test bed TI.

of Algorithm 6.3 without storing all non-dominated candidates is smaller than 0.7 seconds for each number of inputs. When all non-dominated candidates are computed, running times are still at most 0.06 seconds for up to 30 inputs, but increase up to 88 seconds for up to 100 inputs. Hence, computing a split-optimum circuit is very fast, but computing a split-optimum circuit with a good size is slower. For Algorithm 6.3 without storing all non-dominated candidates, we have $\lambda = 3.923$, which matches the running time guarantee of $\mathcal{O}(m^4)$ given in Theorem 6.1.14. We explained in Section 6.1.4 that storing all non-dominated candidates will increase the running time by a factor of at most S^2, where S is the size of the constructed circuit. As Figure 6.10(a) suggests the assumption that the circuit size grows roughly linear in m, when all non-dominated candidates are stored, $\lambda \approx 6$ is as expected.

For instances with fractional arrival times, the average running time of the binary-search extension of Algorithm 6.3 is indeed roughly $\log_2 m$ times the average running time of Algorithm 6.3 with size optimization enabled. Also, $\lambda = 6.517$ is not much higher than for the integral algorithm. As expected, for the linear-search extension of Algorithm 6.3, the average running times are roughly m times the integral running times, and we have $\lambda \approx 7$. In particular, the average running times are much higher than for the binary-search extension. The running time of the explicit extension of Algorithm 6.3 is even higher, and we have $\lambda = 8.714$. As expected, the maximum number of candidates per $f_{i,j,k}$ is much higher for the explicit extension than for the binary extension – e.g., for the instances with 50 inputs, by a factor of 5 up to 40 higher. This number contributes a quadratic factor to the running time; and additionally, in the explicit extension, we need to perform Huffman coding for every merge of 2 candidates.

In our practical application in chip design, see Section 7.3, we use the binary-

	Integral arrival times		Real arrival times		
# inputs	No size opt.	With size opt.	Binary	Linear	Explicit
10	0.000	0.000	0.001	0.001	0.001
20	0.001	0.007	0.019	0.092	0.069
30	0.005	0.054	0.310	1.351	1.281
40	0.017	0.290	1.808	10.287	14.787
50	0.035	1.179	5.884	56.195	99.164
60	0.067	3.739	26.693	224.705	808.890
70	0.147	9.665	78.845	770.667	3221.723
80	0.281	20.185	207.164	2021.854	8972.769
90	0.423	46.051	351.129	4526.578	34365.340
100	0.662	87.701	696.301	10162.449	85267.046
λ	3.923	5.863	6.517	7.215	8.714

Table 6.2: Average running times of Algorithm 6.3 on 10 randomly generated And-Or path instances for each number of inputs. For integral arrival times, we show running times both with and without the size optimization from Section 6.1.4 which comptues all non-dominated candidates. For fractional arrival times, we show the running times of all three extensions of Algorithm 6.3 to fractional arrival times from Section 6.1.3. In the last row, for each algorithm A with average running time $r(A, n)$ for n inputs, we show the value $\lambda(A) = \log_{100/20}\left(\frac{r(A,100)}{r(A,20)}\right)$.

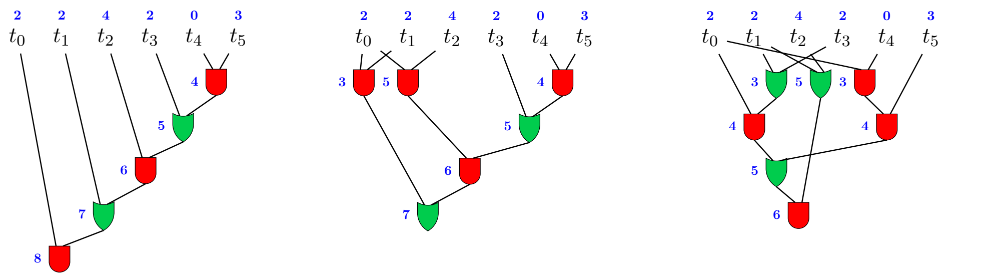
search extension of Algorithm 6.3 to fractional arrival times. We shall see in Section 7.4 that we can certainly afford this running time: Mostly, our instances have only up to 20 inputs, where the average running time of our algorithm is 0.015 seconds or less. Other parts of our flow presented in Section 7.3 consume the majority of the running time, see Table 7.1. As we apply the And-Or path optimization algorithm several hundred times during our flow, the running times of our exact algorithm from Chapter 5 as presented in Table 5.3 are too high for this application, at least if size optimization is enabled.

6.3 Comparison with Exact Algorithm

In this section, we correlate the algorithms of Algorithms 5.1, 6.1 and 6.3. In order to understand the difference between Algorithm 6.1 and Algorithm 5.1, let us consider the structure theorem, Theorem 5.2.9, for the case of extended And-Or paths. For this study, we assume that Conjecture 5.2.11 is fulfilled.

So consider inputs $t = (t_0, \dots, t_{m-1})$ with uniform input arrival times $a(t_0) = \dots = a(t_{m-1}) \in \mathbb{R}$ and gate types $\Gamma = (\circ_0, \dots, \circ_{m-1})$. Assume that $h(t; \Gamma)$ is an extended And-Or path, i.e., its signal partition $\{t_0, \dots, t_{m-1}\} = P_0 \mathbin{+\!\!\!+} \dots \mathbin{+\!\!\!+} P_c$ fulfills $|P_1| = \dots = |P_{c-1}| = 1$ and $|P_c| = 2$. Consider a size-optimum circuit C among all depth-optimum formula circuits for $h(t; \Gamma)$.

For $k \in \{1, 2\}$, let i_k denote the maximum index of any input in S_k°. Without loss of generality, we may assume that $i_2 = m - 1$. By Conjecture 5.2.11, for each $t_j \in S_2^\circ$, we have $i_1 < j$. Consider the circuit $C = C_1 \circ C_2$ arising from applying Algorithm 5.1 to inputs $t_0, \dots, t_{m-1}$, and assume that $\circ =$ And (the case $\circ =$ Or



2 2 4 2 0 3
t_0 t_1 t_2 t_3 t_4 t_5
4
5
6
7
8
2 2 4 2 0 3
t_0 t_1 t_2 t_3 t_4 t_5
3 5 4
5
6
7
2 2 4 2 0 3
t_0 t_1 t_2 t_3 t_4 t_5
3 5 3
4 4
5
6

(a) An AND-OR path with delay 8.

(b) Solution C_U with delay 7 for the instance from Figure 6.12(a) computed by Algorithm 6.3 (page 170).

(c) Optimum solution with delay 6 for the instance from Figure 6.12(a) computed by Algorithm 5.1.

Figure 6.12: An instance where Algorithm 6.3 (page 170) does not compute a delay-optimum solution.

works analogously by dualizing).

We show that C can be constructed via one of the splits (6.1), (6.2) or (6.3). By Observation 6.1.2, Algorithm 6.1 computes a circuit with best delay achievable with these splits, so this would imply that Algorithm 6.1 and, by Proposition 6.1.12, also Algorithm 6.3, are also exact algorithms in the case of uniform arrival times.

Case 1: We have $t_{i_1} \in P_0$.

In particular, in this case, we have $\circ_0 = \text{AND}$. The function $h(t;\Gamma)_{S_1^{\text{AND}}}$ is a symmetric function on a consecutive prefix of P_0, and the function $h(t;\Gamma)_{S_2^{\text{AND}}}$ is an extended AND-OR path arising from $h(t;\Gamma)$ by deleting a consecutive prefix of P_0. Hence, $C = C_1 \wedge C_2$ arises from a symmetric split as in Equation (6.3).

Case 2 We have $t_{i_1} \notin P_0$.

Now, we have

$$f_1 = h(t;\Gamma)_{S_1^{\text{AND}}} = h(t;\Gamma)_{[0:i_1]}$$

and

$$f_2 = h(t;\Gamma)_{S_2^{\text{AND}}} = \bigwedge_{i<i_1, \circ_i = \text{OR}} t_i \wedge h(t;\Gamma)_{[i_1+1:m-1]},$$

i.e., f_1 and f_2 are both extended AND-OR paths. As $t_{i_1} \in S^{\text{AND}}$, we have $\circ_{i_1} = \text{AND}$ and $\circ_{i_1+1} = \text{OR}$. Hence, the number of inputs of f_1 that are not contained in P_0 is even if and only if $\circ_0 = \text{OR}$. Moreover, the inputs of P_0 are contained in f_2 if and only if $\circ_0 = \text{OR}$. Hence, the circuit $C = C_1 \wedge C_2$ arises from an odd split Equation (6.1) if $\circ_0 = \text{AND}$ and an from even split Equation (6.2) if $\circ_0 = \text{OR}$.

Thus, Conjecture 5.2.11 implies the following conjecture.

Conjecture 6.3.1. *For uniform arrival times, both Algorithms 6.1 and 6.3 compute optimum solutions for all extended* AND-OR *paths.*

As by Table 6.1 (page 177), this is statement is satisfied for all AND-OR path instances with up to 109 inputs, we assume this conjecture as well as Conjecture 5.2.11 to be true.

For the case of nun-uniform arrival times, already for 6 inputs, there is an AND-OR path instance with certain input arrival times where Conjecture 5.2.11 is not fulfilled. We show this instance in Figure 6.12(a). In the circuit in Figure 6.12(b) computed by Algorithm 6.3 (page 170), we are not able to connect the input t_2 with arrival time 4 to the output via only two gates because the AND-OR path on the last 3 inputs has delay 5. In the circuit in Figure 6.12(c) which is a possible output of Algorithm 5.1, t_2 has depth 2 because this realization can cut out the inputs t_1 and t_2 from the AND-OR path, which is not possible in our algorithm.

Indeed, it is easy to verify that the function computed by the right predecessor of $\operatorname{out}(C)$ in Figure 6.12(c) is the generalized AND-OR path $t_0 \wedge (t_1 \vee t_3 \vee (t_4 \wedge t_5))$, which is not an extended AND-OR path. Furthermore, the partition of the same-gate signals S^{AND} of $h(t;\Gamma)$ in this case is $S_1^{\text{AND}} = \{t_0, t_4, t_5\}$ for the left sub-circuit and $S_2^{\text{AND}} = \{t_2\}$ for the right sub-circuit.

Chapter 7

BonnLogic: A Logic Restructuring Flow

In this chapter, we consider a practical application of And-Or path optimization: logic restructuring in VLSI (very large scale integration) design, i.e., chip design. The Research Institute for Discrete Mathematics at the University of Bonn develops a VLSI software suite called BonnTools in a long-term cooperation with IBM, see, e.g., Korte, Rautenbach, and Vygen [KRV07] or Held et al. [Hel+11]. Hundreds of VLSI chips have been designed with the BonnTools, among those the latest POWER and mainframe processors.

The BonnTools contain optimization algorithms for most important sub-problems arising in chip design. In particular, there are several timing optimization algorithms which ensure that all signals on a chip meet their respective deadlines. One of these timing optimization tools is BonnLogic, a tool that revises the logical structure of timing-critical parts of a chip during the VLSI design flow. An earlier version of BonnLogic was published by Werber, Rautenbach, and Szegedy [WRS07] and, with more details, by Werber [Wer07]. The current version has been published previously in Brenner and Hermann [BH20].

In Section 7.1, we give a brief introduction into VLSI design with a focus on timing optimization. An overview of the previous work on logic optimization in chip design is given in Section 7.2. In Section 7.3, we describe BonnLogic in detail, and in Section 7.4, we demonstrate its effectiveness and efficiency in experiments on recent 7nm real-world instances.

7.1 VLSI Design

Physically, a VLSI chip consists of multiple layers: At the bottom, there is the placement layer where the **transistors** are located. Transistors are connected by **wires** which run in several wiring layers and via layers. Each wiring layer contains axis-aligned wires, usually all running in the same direction. Different wiring layers are connected by via layers.

The logic functionality of a computer chip can be modeled by a **netlist** consisting of cells, pins, and nets: The external connection points of a chip are called **primary input** and **output pins**. Each primary output pin models a Boolean function that depends not only on the primary input pins, but also on bits from earlier computations, which are stored in **register cells**. **Logic cells** implement elementary Boolean functions such as And and Inv which together model the Boolean functions

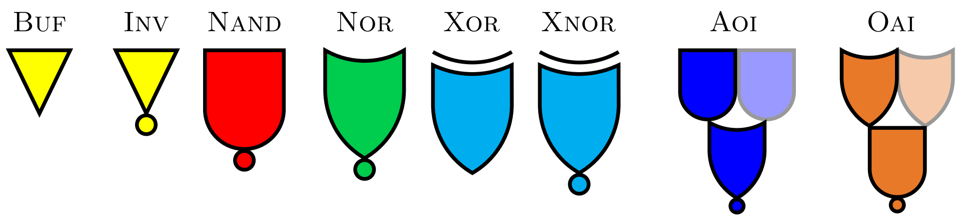


Figure 7.1: A typical CMOS library. Buffer gates denoted by Buf compute the identity function, inverter gates have already been introduced in Figure 2.2. Often, there are no And and Or gates, but only their inverted counterparts Nand and Nor, with up to 4 inputs. Additionally, there are Xor and Xnor gates, where Xnor is the inverted counterpart of Xor with up to 4 inputs. Furthermore, there are two-level gates which are decomposed of two levels of And and Or gates, plus an inverter. E.g., given inputs $a_0, a_1, b_1, b_2 \in \{0, 1\}$, the Aoi21 gate computes the function $\overline{(a_0 \wedge a_1) \vee b_1}$, and the Oai22 gate computes $\overline{(a_0 \vee a_1) \wedge (b_1 \vee b_2)}$.

computed by the chip. Each cell has a set of **pins** for its connection to other cells, or to primary input and output pins. Each such connection is modeled by a **net**, a set of pins that need to be electrically connected by **wires**. Each net contains a single **driver** and arbitrarily many **sinks** and distributes the electrical signal of the driver to all its sinks. A driver may be a primary input pin or the **output pin** of a cell, and a sink may be a primary output pin or the **input pin** of a cell.

Note that when there are no register cells, the logic functionality of a netlist can be modeled by a circuit (Definition 2.2.2). On a chip, the set of Boolean functions for which logic cells are available – which we called a basis in Definition 2.2.1 – is called a **library**, and elements of the library are called **gates** or **gate types**. Nowadays' transistors are CMOS transistors, i.e., complementary metal–oxide–semiconductor transistors. Such devices inherently invert their input signals, so inverting gates can be built using less transistors compared to non-inverting gates. Many CMOS libraries do not contain And and Or gates, but only their inverted counterparts. A typical CMOS library is shown in Figure 7.1.

Apart from the gate type, a logic cell also captures physical data (a physical shape, a physical location on the chip area, locations for its pins), internal transistors, and electrical properties. Hence, there are numerous different logic cells implementing the same Boolean function. Usually, for one gate, there are logic cells with different **gate sizes**, i.e., transistor sizes, and **V_t levels**, i.e., transistor voltages, which have a huge impact on the timing properties of a logic cell. In different regions of a computer chip, different gate sizes are used in order to trade off speed with area or power consumption.

The standard industrial method for analyzing the timing behaviour of a netlist is **static timing analysis**, which goes back to Hitchcock, Smith, and Cheng [KC66; HSC82]. In contrast to Definition 2.3.2, in this context, the term **delay** refers to the time difference between two events. For example, the time a signal needs to traverse a cell or a wire is called **cell delay** or **wire delay**, respectively. Assuming that cell delay and wire delay are given by black-box functions, and assuming that we know the **arrival time** of signals at each primary input pin and the output pin of each register cell, we can propagate arrival times through the netlist (actually, through a so-called timing graph which is a refinement of the netlist, but this is not important for our purpose). At the primary output pins and the input pins of register cells, there might be a **required arrival time** indicating the latest acceptable arrival time. Required arrival times can be propagated backwards through the netlist. At a pin, the difference between required arrival time and arrival time is called **slack**. The slack is non-negative if and only if the signal arrives in time.

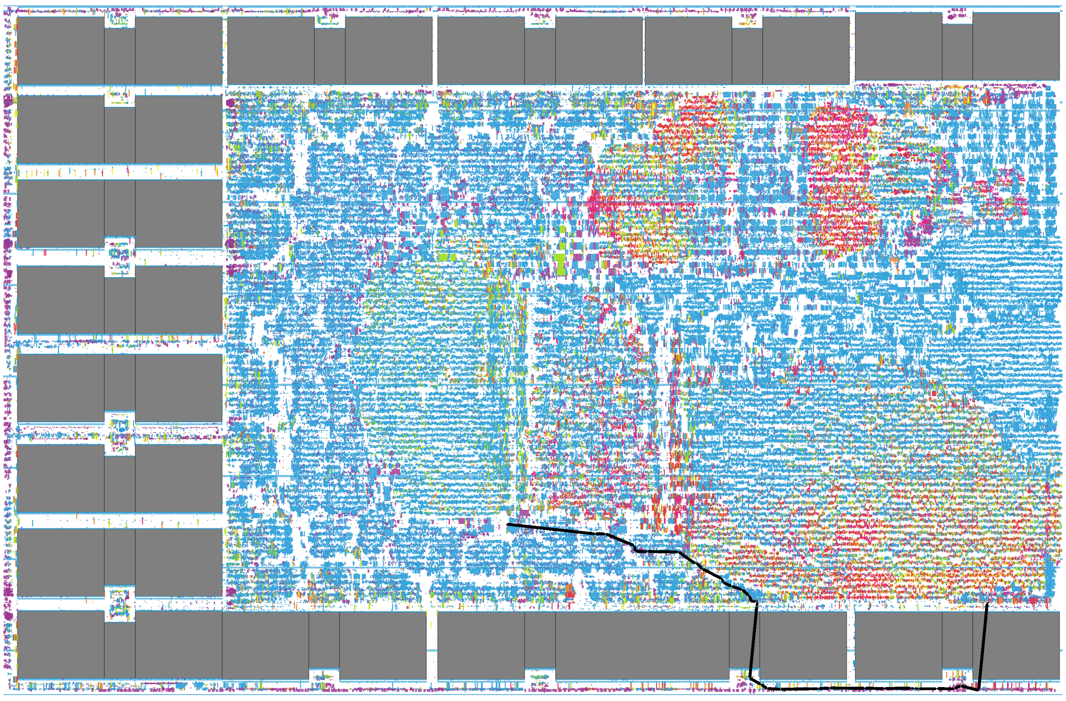
Wire delay and cell delay are always approximations as an exact analysis of the timing behaviour of real hardware is impossible. Delay depends on numerous parameters, e.g., physical distances, electrical capacitances and resistances of cells and wires, and **slews** (the time the voltage function needs to switch from 10% to 90%). E.g., cells with larger transistor sizes can drive higher capacitive loads, i.e., longer wires or more successors.

There are various delay models that differ in accuracy, complexity and simplifying assumptions. A wire delay model that is widely used in timing optimization because it is quite precise and can be computed efficiently is the **Elmore delay model** by Elmore [Elm48]. Here, the netlist is modeled as an electrical network of resistances and capacitances, and the delay of a wire for a net with only one sink is asymptotically quadratic in the length of the wire. Hence, long connections can be sped up by inserting buffers at regular distances in order to make the total wire delay grow only linearly. With more computational effort, capacitances can be modeled more accurately by approximately solving differential equations as in the **RICE evaluation** by Ratzlaff and Pillage [RP94].

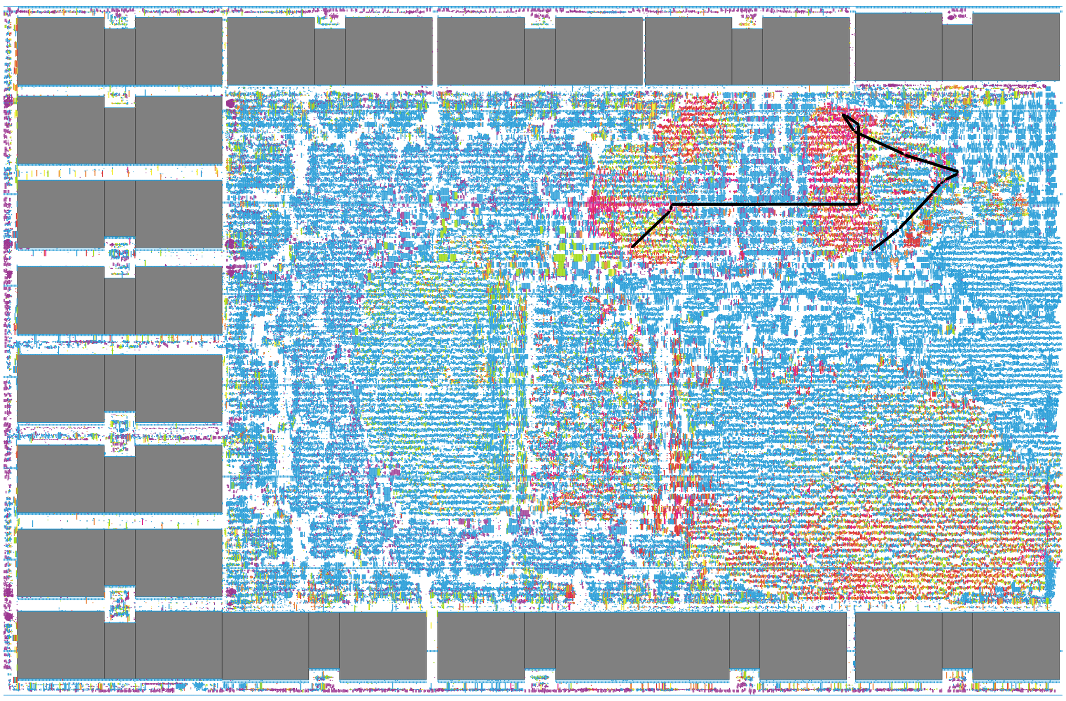
Estimating the delay on the most critical parts of a chip is easier. Here, we know that wires will be buffered optimally, so it is justified to assume that the delay of a wire grows linearly with its physical length. Additionally, we may assume that for each gate, the fastest gate size and V_t level will be chosen, and that the number of successors of cells and slews are low. Hence, the delay of logic cells usually does not vary much in comparison to the differences in locations and arrival times. Cell delay can thus be approximated by a constant (or a different constant for each gate). This simple delay model is a **virtual timing model**, and it yields reasonably good approximations in timing-critical regions of a computer chip (see, e.g., Otten [Ott98] and Alpert et al. [Alp+06]).

Typical measures for the timing criticality of a computer chip are **worst slack**, i.e., the worst slack of any path on the chip, and **sum of negative endpoint slacks**, i.e., the sum over the negative slacks of all paths between timing endpoints (register cells or primary inputs / outputs).

In Figure 7.2, we plot two similar netlist for the same computer chip, called i6. This chip is a 7nm chip that is currently designed by IBM, and the netlist is not yet in a final state that is going into production. The grey blocks are **macros**, i.e., mostly rectangular areas of the chip which are blocked and cannot be changed. Macros may contain regular structures of register cells or, hierarchically, another computer chip. Thus, macros also have pins and incident nets. The chip i6 itself is a macro of another computer chip on a higher hierarchical level. The colored objects are the cells. The color of a cell indicates its timing criticality regarding the RICE delay model [RP94] on a scale from blue over green (both mean positive slack) to yellow, red, and violet (negative slack). The black line shows the most timing-critical path of either netlist. i6 is one of the designs on which we examine the behaviour of our logic restructuring tool BonnLogic, see Table 7.1. Figures 7.2(a) and 7.2(b) show



(a) Initial state of i6 with worst slack -38.9 ps and sum of negative endpoint slacks -19.7 ns.



(b) State of i6 after application of BonnLogic to the instance from Figure 7.2(a) with worst slack -23.9 ps and sum of negative endpoint slacks -13.6 ns.

Figure 7.2: Chip i6 from Table 7.1 before and after application of BonnLogic. The large grey blocks are macros. The colored object are the cells, and colors are chosen based on timing criticality, where blue cells are most uncritical and violet cells are most critical. The black line indicates the most timing-critical path.

the netlist before and after application of BonnLogic. Although on a global view, the netlist and slack distribution looks very similar, worst slack and sum of negative endpoint slack improve significantly.

As modern computer chips contain billions of transistors, the VLSI design process is divided into many different steps. At first, the abstract logical description of a chip is modeled in a hardware description language (HDL), see Mermet [Mer93] for an overview. The **logic synthesis** step (see also Section 7.2) turns the abstract logic specification of a chip into a netlist.

During **physical design**, a **placement** (i.e., physical locations for all cells) and a **routing** (i.e., realizations of all nets by electrical wires on the wiring and via layers) is computed for the netlist. See Nam and Cong [NC07] and Markov, Hu, and Kim [MHK15] for a survey on placement algorithms, Gester et al. [Ges+13] for detailed information on routing and Alpert, Mehta, and Sapatnekar [AMS08] for a good overview on both topics. During placement and routing, it is important to ensure that the timing requirements of all signals are met and power consumption and manufacturing costs are low (which usually corresponds to a low area consumption by cells and wires). There are also dedicated timing optimization algorithms. Classical timing optimization tools are **gate sizing**, **V_t assignment** and **buffering** algorithms, or mixtures of these. Timing optimization algorithms that change the logical structure of the netlist are called **logic optimization** algorithms. We discuss logic optimization in detail in the next section. For a comprehensive overview on timing optimization see Sapatnekar [Sap04], Held [Hel08], or Schorr [Sch15].

After physical design, the **layout verification** step tests whether the physical layout works correctly. At the end, the chip is produced in the **fabrication** step.

7.2 Previous Work on Logic Optimization

In general, finding a logically equivalent implementation of a given circuit with, say, minimum depth is an NP-hard problem, see Remark 2.3.10.

A sub-problem which is often considered both in literature and in practice is the **two-level logic minimization** problem. Here, the solution space is restricted to disjunctive normal forms, see Equation (2.9). A classical two-level optimization problem asks for a disjunctive normal form with minimum size. The corresponding decision problem is NP-complete if the input is given as a truth table, and it is even Σ_2^P-complete if the input is given by a disjunctive normal form (see the survey of Umans, Villa, and Sangiovanni-Vincentelli [UVS06]). For the definition of the class Σ_2^P, which follows NP in the polynomial hierarchy, see Garey and Johnson [GJ79], Section 7.2). Roughly, a problem is in Σ_2^P if it can be solved in polynomial time assuming that an oracle from the class NP may be used.

The first exact algorithm for the two-level optimization problem is the Quine-McCluskey approach (see, e.g., Crama and Hammer [CH11]) which has been developed by Quine [Qui52] and extended by McCluskey [McC56]. The algorithm generates all prime implicants and then finds a subset that suffices for a disjunctive normal form via set covering. Many practical algorithms for two-level optimization are based on the Quine-McCluskey approach, but avoid the generation of all prime implicants, e.g., the Espresso algorithms (Rudell and Sangiovanni-Vincentelli [RS87] and McGeer et al. [McG+93]) and the Scherzo tool (Coudert [Cou94]).

In practice, logic synthesis is divided into two stages: First, a technology-independent logic description is computed and optimized, e.g, a description by a circuit containing only And2 and Inv gates (called AIG), or containing sums of products (called SIS). Here, **multi-level logic minimization**, the generalization of two-level optimization to an arbitrary number of levels, is applied. There are exact integer programming methods for the general problem, but they are only viable for very small circuits, (see Muroga [Mur91]). Heuristic approaches are described, e.g., in the surveys by Devadas, Ghosh, and Keutzer [DGK94] and Hachtel and Somenzi [HS06]. The two main ideas here are to rewrite Boolean formulae using algebraic operations and to examine so-called "don't cares" – truth value assignments at gates that can never occur. These ideas are also the basis for industrial logic synthesis tools like BooleDozer (see Stok et al. [Sto+96]), ABC (see Brayton and Mishchenko [BM10]), and the Synopsys Design Compiler (see [Syn20]).

These tools also cover the second step in logic synthesis, i.e., computing a technology-dependent description that is both fast and compact making use of all gates in the library. This step is called **technology mapping**. Although technology mapping is usually only allowed to change the given circuit locally, the established versions of technology mapping for area optimization are shown to be NP-hard on general circuits by Keutzer and Richards [KR89].

But on tree-like circuits where the underlying undirected graph contains no cycles, the technology mapping problem for, e.g., area optimization under delay constraints, is solvable in polynomial time by dynamic programming (see Chaudhary and Pedram [CP92]). Hence, the netlist is often partitioned into tree-like sub-circuits, where technology mapping is performed independently. This technique goes back to Keutzer [Keu87], whose algorithm was the starting point for modern technology mapping. Technology mapping usually tries to cover the input circuit by graphs representing the available gate types. Thus, there is the inherent problem that the solution significantly depends on the input circuit. There were multiple enhancements of Keutzer's algorithm over the years, which in particular try to overcome this problem and to extend technology mapping to general circuits. Often, they are based on so-called Boolean matching, see, e.g., Mailhot and Di Micheli [MD93] and Chatterjee et al. [Cha+06]. Here, as a preprocessing step, a hash table is computed which contains the truth table of the Boolean function for each library gate. During technology mapping, the truth table for sub-circuits is compared to the stored Boolean functions in order to determine possible replacement gates.

When area is ignored and only delay is optimized, the technology mapping problem is solvable in polynomial time on general circuits, see Kukimoto, Brayton, and Sawkar [KBS98].

The BonnTools software suite contains the technology mapping algorithm by Elbert [Elb17]. This is a fully polynomial-time approximation scheme (FPTAS) for optimizing a trade-off of delay and size on circuits with a single output and a constant number of gates with more than one successor. More details about this algorithm are given in Section 7.3.2.

During logic synthesis, no physical information has been computed yet, so it is not yet known which parts of the chip may turn out to be timing-critical during physical design. As a tight production schedule often does not allow iterating the whole design process, starting with a logic synthesis step based on timing-information, there are logic optimization algorithms especially designed to run during physical design.

Many approaches work on arbitrary Boolean functions and hence can only replace sub-functions of constant size by alternative realizations (see e.g., Stok et al. [Sto+96], Cortadella [Cor03], Trevillyan et al. [Tre+04], Mishchenko et al. [Mis+11], and Amarú et al. [Ama+17]). Here, the new solution is logically correct by construction, but a more global logic restructuring is hardly possible. Plaza, Markov, and Bertacco [PMB08] apply logic modifications (similar to cloning methods) after placement to improve the delay of critical paths that have been placed with detours. But this approach is also restricted to local changes because the logical equivalence of different implementations is shown by enumerating possible input truth assignments.

7.3 Flow Description

Logic synthesis is one of the first steps in the design process and for the following steps, the logical description typically remains fixed. However, during physical design, it may turn out that the chosen implementation of the logic functionality was not the best choice, e.g., with respect to placement or timing. Now it would be desirable to find a better suited logically equivalent representation. As described in Section 7.2, other logic optimization techniques used for fixing timing problems during physical design work only locally on small fractions of the netlist, and often do not have provable approximation guarantees. In contrast to these methods, BonnLogic can resynthesize combinatorial paths of arbitrary length and thus resolve more complex timing problems.

BonnLogic is a timing optimization flow that is used in several steps of IBM's physical design flow: for assertion generation before placement, i.e., generation of (required) arrival times for certain pins, for optimizing timing late in the pre-routing physical design flow, and for executing engineering change orders (ECOs). Here, we focus on the second application.

In a global view, timing optimization needs to be done in a careful way in order to find good trade-offs between signal speed and area / power consumption. Still, the most timing-critical part of a chip needs to be well-optimized as it is the limiting factor for the clock frequency of a computer chip. There are other timing and in particular logic optimization algorithms in the BonnTools that optimize timing globally. BonnLogic, however, is a tool for optimizing the most critical paths.

BonnLogic was originally developed by Werber, Rautenbach, and Szegedy [WRS07] and has been improved significantly during the last years, see a previous publication by Brenner and Hermann [BH20]. The idea of BonnLogic is to improve worst slack by iteratively restructuring the most critical path. As observed by Werber, Rautenbach, and Szegedy [WRS07] (see also Section 7.3.1), optimizing a path can be reduced to optimizing an And-Or path, cf. Definition 2.5.1. Hence, the essential component of BonnLogic is an And-Or path optimization algorithm – in the original version by Werber, Rautenbach, and Szegedy [WRS07], this is the algorithm from Rautenbach, Szegedy, and Werber [RSW06]; and in the current version, this is Algorithm 6.3, which has a much better approximation guarantee (cf. Theorem 6.1.14 and Table 2.2) and performs much better on most instances, see Section 6.2.

As BonnLogic is used during physical design, placement and timing information need to be taken into account during optimization. In Section 7.3.1, we hence adapt the simple delay model used in Algorithm 6.3 to respect placement, buffering and gate sizing effects. As we do not fully account for different kinds of gates or different gate sizes that might be available, our framework involves a technology mapping step (cf. Section 7.3.2) and powerful gate sizing and buffering routines (cf. Section 7.3.3). As buffering and gate sizing perform accurate delay computations, which are very complex, this is by far the most time-consuming part of our flow and we need to

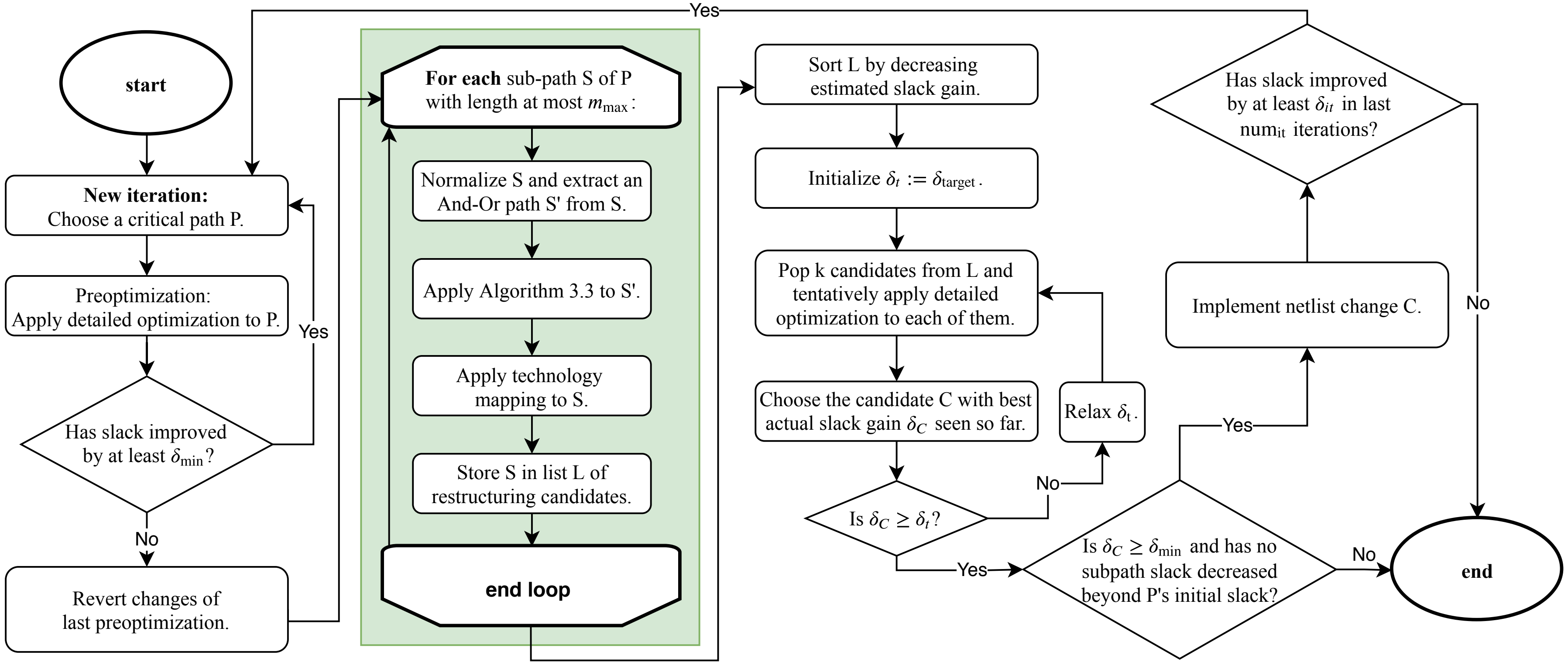

start
New iteration: Choose a critical path P.
Preoptimization: Apply detailed optimization to P.
Has slack improved by at least $\delta_{\min}$?
Yes
No
Revert changes of last preoptimization.
For each sub-path S of P with length at most $m_{\max}$:
Normalize S and extract an And-Or path S' from S.
Apply Algorithm 3.3 to S'.
Apply technology mapping to S.
Store S in list L of restructuring candidates.
end loop
Sort L by decreasing estimated slack gain.
Initialize $\delta_t := \delta_{\text{target}}$.
Pop k candidates from L and tentatively apply detailed optimization to each of them.
Choose the candidate C with best actual slack gain δ_C seen so far.
Is $\delta_C \geq \delta_t$?
No
Relax δ_t.
Yes
Is $\delta_C \geq \delta_{\min}$ and has no subpath slack decreased beyond P's initial slack?
Yes
No
Implement netlist change C.
Has slack improved by at least δ_{it} in last num_{it} iterations?
Yes
No
end

Figure 7.3: Flow chart for our logic optimization framework (cf. Section 7.3) with the path restructuring step in green.

limit its usage.

We now outline our logic optimization flow in detail before describing different sub-steps in the subsequent sections. The flow is illustrated in Figure 7.3. All parameters mentioned are chosen technology-dependently, but can also be modified by users.

BonnLogic iteratively optimizes the worst slack of the currently most timing-critical combinational path until the overall worst slack does not improve significantly anymore. A single **iteration** works as follows:

Let P denote a most critical path. During a **preoptimization** step, we first try to improve the slack of P without changing its logical structure in order to diminish disruptions. To this end, we apply **detailed optimization** to P as described in Section 7.3.3. If a threshold slack improvement of δ_{min} is exceeded, we keep the changes imposed by preoptimization and start the next iteration.

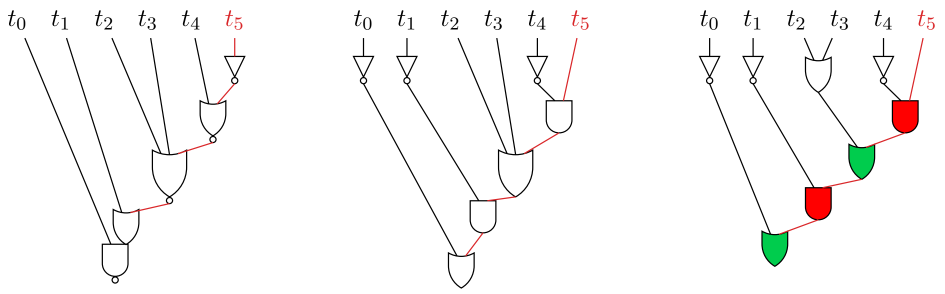
Otherwise, we discard the preoptimization's changes and perform the **path restructuring** step (central, green part of Figure 7.3). This step works using internal data structures and internal virtual delay models; the netlist is not changed before detailed optimization (Section 7.3.3). Due to the inaccuracy of our timing model, we consider the possibility to optimize any sub-path S of P up to a maximum length of m_{max}. First, we apply a **normalization** step (Section 7.3.1) in order to extract an And-Or path S' from S on which we run Algorithm 6.3 to determine the global structure of the replacement circuit. Then, the **technology mapping** routine from Elbert [Elb17] (see also Section 7.3.2) locally modifies S to benefit from all gates available in the library. After having optimized all sub-paths of P, we store all restructuring possibilities in a list L, sorted by decreasing estimated slack gain.

For only the most promising fraction of restructuring options, we apply the time-consuming **detailed optimization** (cf. Section 7.3.3): First, we tentatively apply detailed optimization to the topmost k candidates in L. If the actual slack gain of the best solution exceeds δ_{target}, we choose this solution; otherwise, we iteratively decrease δ_{target} by a fixed value and try out the next k candidates in L until we reach δ_{target} or L is empty. Afterwards, we choose the restructuring candidate C with best actual slack gain δ_C for P among all detailed-optimized solutions. This way, we usually apply detailed optimization to only a few instances, but still find a good restructuring option. If $\delta_C \geq \delta_{\text{min}}$ and if no side path slack has worsened beyond the initial slack of P, we implement this netlist change, possibly retaining parts of P needed for side outputs. If the change is implemented and the slack gain over the last num_{it} iterations exceeds a threshold δ_{it}, we start the next iteration; otherwise, we stop.

Note that this is a simplified flow description. E.g., in practice, we optimize the second critical path or the most critical path between register cells when P cannot be further optimized.

7.3.1 Delay Model and Normalization

Our And-Or path optimization algorithm, Algorithm 6.3, expects as an input an alternating path of And2 and Or2 gates with prescribed input arrival times, and assumes that gates have a unit delay and connections do not impose any delay (see the delay model defined in Definition 2.3.2). However, the most critical path P contains arbitrary gates with varying delays, and the physical locations of the path inputs might be far apart, inducing undeniably high wire delays even after buffering. A **normalization** step thus transforms P into a piece of netlist whose core part is an And-Or path with appropriately modified input arrival times.



(a) Sub-path S containing an inverter, an OR2, an OR3, and an OAI gate. Input t_5 is most critical.

(b) Netlist resulting from Figure 7.4(a) by pushing inverters away from the critical path.

(c) Netlist resulting from Figure 7.4(a) by performing Huffman coding for the OR3 gate.

Figure 7.4: Normalizing a sub-path S (Figure 7.4(a)) of the critical path P. In the resulting netlist in Figure 7.4(c), the extracted AND-OR path S' is colored. Critical connections are drawn in red.

Before explaining our normalization, we revise our timing model. As we work on the most critical path, the buffering routine applied in Section 7.3.3 will compute delay-optimum solutions. Hence, it is realistic to assume a virtual timing model (see Alpert et al. [Alp+06] and Otten [Ott98]), which is briefly introduced in Section 7.1. Thus, we may assume a linear wire delay and estimate the wire delay between two physical positions p_1 and p_2 by $d_{\text{dist}} \cdot ||p_1 - p_2||_1$ for a constant $d_{\text{dist}} \in \mathbb{R}$. The traversal time through a gate is approximated by a constant $d_{\text{gate}} \in \mathbb{R}$. The constants d_{gate} and d_{dist} are chosen based on an analysis of typical values on the respective design. In particular, d_{dist} is computed by analyzing long buffer chains as described by Bartoschek et al. [Bar+06]. As on the critical path, there are rather low fan-outs and slews, the delay of gates with different types and sizes still varies, but not much in comparison to the differences in arrival times. Hence, assuming a realistic constant gate delay suffices to determine the logical structure of the circuit.

Since we work on the most timing-critical part of the design, we place the circuit C computed by Algorithm 6.3 such that each path is embedded delay-optimally accepting possible netlength increases. For instance, placing all gates at $l(\text{out}(C))$ would ensure this, implying that each path from an input t_i to $\text{out}(C)$ has a wire delay of $d_{\text{dist}} \cdot ||l(t_i) - l(\text{out}(C))||_1$, where l indicates physical coordinates on the chip. In fact, we choose a placement that is netlength-optimum among all delay-optimum placements: We determine $l(\text{out}(C))$ based on its successors in the netlist and place each gate at the median position of its predecessors and $\text{out}(C)$, see Teichmann [Tei13]. Thus, the delay of C is

$$\max_{Q\colon t_i \rightsquigarrow \text{out}(C)} \Big\{ a(t_i) + d_{\text{dist}} \cdot ||l(t_i) - l(\text{out}(C))||_1 + d_{\text{gate}} \cdot |Q| \Big\},$$

where the maximum ranges over all directed paths Q in C from any input t_i to $\text{out}(C)$. Applying Algorithm 6.3 with modified arrival times

$$a'(t_i) := \frac{1}{d_{\text{gate}}} \Big(a(t_i) + d_{\text{dist}} \cdot ||l(t_i) - l(\text{out}(C))||_1 \Big)$$

for all $i \in \{0, \ldots, n-1\}$ hence yields a circuit with optimum wire delay with respect to physical locations.

Now, we can describe our normalization of a sub-path S of P with the help of Figure 7.4. Let x denote the most critical input of S. We replace each gate in S by a representation using And2, Or2 and Inv gates (in Figure 7.4, this can be derived easily from Figure 7.4(a) and is not shown explicitly). This does not necessarily yield a path, but we can recover the original critical path by following the signal flow of x, obtaining a path S'. By applying De Morgan's transformations (see also Equation (2.6)) in reverse topological order, we remove inverters from S' while possibly adding inverters at the inputs of S' (see Figure 7.4(b)). Note that this increases the delay of any path in S by at most the delay of one inverter. Now, S' is a generalized And-Or path (see Definition 2.5.5), and for its optimization, we use the following simple translation to And-Or paths: We use Huffman coding (see Theorem 2.3.21) on chains of And gates (and Or gates) in S' to move less critical gates into $S \backslash S'$ in order to make S' an And-Or path (see Figure 7.4(c)). As we also need to incorporate physical distances and gate and distance delay during Huffman coding, we use our virtual timing model as defined above. This way, S' becomes an And-Or path that – with input arrival times a' – can be passed to Algorithm 6.3.

7.3.2 Technology Mapping

After computing a circuit using Algorithm 6.3, we invoke a **technology mapping** step. Its purpose is to change the newly created circuit locally to improve worst slack and the physical area occupied by gates by making use of all gates available on the design. We use the dynamic programming algorithm by Elbert [Elb17] which applies Boolean matching (see also Section 7.2) to cover the input circuit by graphs representing the available gate types. This algorithm works with a virtual delay model that generalizes the model described in Section 7.3.1: Gates have pin-dependent delay parameters to incorporate that the time needed to traverse a gate varies from pin to pin, and for each gate type, the pin-dependent gate delay values are computed separately. Wire delay is estimated depending on the optimum layer for the given distance.

Elbert's algorithm computes an optimum technology mapping with respect to any fixed trade-off of arrival time and number of gates, but the running time grows exponentially in the number l of gates with more than one successor. In our application, l is usually very small, hence we can afford this running time (cf. the running time discussion at the end of Section 7.4). For constant l, Elbert [Elb17] also provides a fully polynomial-time approximation scheme. On general circuits, computing a size-optimum technology mapping is NP-hard by Keutzer and Richards [KR89].

7.3.3 Detailed Optimization

The application of our logic restructuring flow in a late stage of physical design implies that the input netlist will already be highly optimized with respect to timing. Most importantly, each gate will have an appropriate gate size and V_t level, long connections will have a proper layer assignment and will be well-buffered. Hence, in order to construct competitive replacement logic, these effects need to be regarded.

As a consequence, depending on the actual stage of the design, our **detailed optimization** step invokes powerful buffering, layer assignment and gate sizing routines. In all these steps, slacks are computed using the timing engine IBM EinsTimer. When applied late in physical design, EinsTimer uses the RICE delay model by Ratzlaff and Pillage [RP94]. This is much more accurate than our virtual delay models used in Sections 7.3.1 and 7.3.2, but also much more time-consuming. Hence, we must control the detailed optimization effort very strictly.

When used in late physical design, we apply the gate sizing routine by Held [Hel09] on all gates touched by logic optimization. This is a preparation for the following buffering step as it is important that during buffering, slacks can be estimated accurately. For buffering, we use the buffering tool with an integrated layer assignment by Bartoschek et al. [Bar+09] on all nets in a small neighborhood. This regards that the timing behavior also might have changed at gates we have not touched in optimization. After buffering, we apply gate sizing again, in particular on newly inserted buffers.

As we work on the most critical fraction of the design, V_t assignment can be done conveniently by using the fastest V_t levels available.

An incremental placement legalization makes sure that the placement remains legal throughout all netlist changes.

7.4 Computational Results

We examine the behavior of BonnLogic on real-world IBM chips in a late, prerouting stage of physical design, which is our main industrial application. Table 7.1 shows results on 8 recent 7nm instances, i1 up to i8, and a single older instances from the 22nm technology, i9.

For each chip, we show two rows: The 'init' row displays the state of the chips before BonnLogic is applied: a timing-driven placement has been computed using the algorithm by Brenner et al. [Bre+15], followed by various timing optimization steps, among those our buffering and gate sizing sub-routines (see Section 7.3.3, Held [Hel09] and Bartoschek et al. [Bar+09]). The initial netlist cannot be improved any further by classical timing optimization. The 'BL' (BonnLogic) row shows results after applying our logic optimization flow to this netlist. Here, the IBM timing EinsTimer engine uses the RICE delay model (see Ratzlaff and Pillage [RP94]) to evaluate timing, both during optimization and for our statistics.

For measuring the impact of BonnLogic on timing, we display worst slack (WS) and sum of negative endpoint slacks (SNES). On instance i9, which is the only 22nm instance, worst slack and sum of negative endpoint slacks improve vastly, which is the case as on this chip, the netlist contains a large, dense connected component of logic cells which is timing-critical. On recent designs, this is rarely the case, and path delays are often dominated by wire delay, so BonnLogic cannot improve worst slack so easily. Still, on most of the 7nm instances, we can improve both worst slack and sum of negative endpoint slacks significantly in comparison to the initial state of the netlist.

We see that our timing improvements do not disrupt global objectives: number of gates, area, netlength, and routability are barely effected by BonnLogic. Here, to check routability, we use the ACE5 estimate by Wei et al. [Wei+12], the average congestion of the 5 % most congested resources, weighted by usage, computed using the global router by Müller, Radke, and Vygen [MRV11].

Figure 7.2 (page 194) depicts the chip i6 before and after application of BonnLogic. The overall structure of the chip as well as its slack distribution look similar in both pictures. However, the amount of cells that are violet (i.e. the most timing-critical), reduces drastically on the right half of the chip. In particular, the path that is most timing-critical in the initial netlist from Figure 7.2(a) has improved by at least 15 ps and is now not timing-critical anymore. In Figure 7.2(b), the most critical path lies in a completely different region of the chip.

Our program was implemented in C++, and all tests were executed on a machine

Unit	Run	WS [ps]	SNES [ns]	# Gates	Area	Netlength	ACE5	T_{total} [s]	T_{AOP} [s]	# Calls	Max. # inputs
i1	init	−107.0	−26.1	22 412			83%				
	BL	−103.9	−26.0	22 431	+0.01 %	+0.00 %	82%	409	0.171	1650	8
i2	init	−14.1	−1.7	38 048			93%				
	BL	−14.0	−1.6	38 067	+0.02 %	+0.00 %	93%	50	0.017	132	7
i3	init	−65.0	−67.4	64 230			97%				
	BL	−53.3	−57.2	64 249	+0.04 %	+0.09 %	96%	140	0.083	914	7
i4	init	−16.9	−1.1	78 193			110%				
	BL	−2.5	−0.1	77 851	-0.28 %	-0.14 %	110%	230	0.166	1528	8
i5	init	−173.6	−335.4	212 210			94%				
	BL	−151.8	−332.9	212 236	+0.01 %	+0.01 %	94%	306	0.259	2178	8
i6	init	−38.9	−19.7	268 473			87%				
	BL	−23.9	−13.6	268 336	-0.00 %	+0.03 %	88%	272	0.342	1254	13
i7	init	−69.3	−182.8	274 723			95%				
	BL	−55.1	−168.9	274 863	+0.03 %	+0.02 %	95%	400	0.116	984	8
i8	init	−124.5	−656.3	332 695			92%				
	BL	−115.5	−640.9	332 787	-0.00 %	+0.02 %	92%	253	0.103	724	5
i9	init	−396.4	−101.8	379 707			95%				
	BL	−216.6	−90.8	379 931	+0.02 %	+0.04 %	95%	1 019	1.998	16 362	7

Table 7.1: Performance of our logic restructuring framework on 8 recent 7nm real-world instances and an older 22nm real-world instance, *i*9.

with two Intel(R) Xeon(R) CPU E5-2667 v2 processors, using a single thread. The column T_{total} shows the total running time of our flow, which is usually a few minutes. Even on instance i9, where slack improvements are very high, BonnLogic only runs 16 minutes. In the last three column, we show the total running time T_{AOP} of all calls to our And-Or path optimization algorithm, Algorithm 6.3, (including normalization), the number of calls to this algorithm and the maximum number of inputs of the And-Or path optimized in any of these calls.

Note that the total running time BonnLogic is roughly proportional to the number of calls to Algorithm 6.3. Still, the running time of normalization and Algorithm 6.3 is below 0.4 seconds for instances i1 up to i8, and only 2 seconds for i9. Also, the running time of the technology mapping step (which is not displayed in the table) is below 4 seconds for instances i1 up to i8, and roughly 18 seconds for instance i9.

However, the number of And-Or path instances considered also is proportional to the number of calls to detailed optimization, and this step dominates the running time of BonnLogic. In particular, the gate sizing step takes much time because it performs many expensive queries to EinsTimer. When a more efficient, but less accurate timing model than RICE timing is used in EinsTimer, the running time of detailed optimization is much lower.

The maximum number of inputs of any And-Or path which is optimized during BonnLogic is often $7-8$, and 13 for instance i6.

Results on publicly available benchmarks are not presented due to the lack of comparable results. For instance there is the EPFL combinational benchmark suite by Amarú, Gaillardon, and De Micheli [AGD15], which provides a set of circuits designed to challenge logic optimization tools. However, the only published results on for this benchmark set optimize LUT-6 depth, not circuit depth. Here, each gate is implemented as a look-up table, and for every Boolean function with up to 6 inputs, a gate is available. LUT-6 gates are used in FPGA (field-programmable gate array) design, see, e.g., Vemuri, Kalla, and Tessier [VKT02].

Chapter 8
Faster Linear-Size Adder Circuits

In this chapter, we consider the Adder Optimization Problem. Recall from Definition 2.4.4 that, given n input pairs $p_0, g_0, \dots, p_{n-1}, g_{n-1}$, we write A_n to denote an adder circuit on n input pairs, and, for $i \in \{1, \dots, n\}$, we write $\operatorname{out}_i(A_n)$ to denote its output computing the carry bit

$$c_i = g_{i-1} \vee \left(p_{i-1} \wedge \left(g_{i-2} \vee \left(p_{i-2} \wedge \left(g_{i-3} \vee \left(p_{i-3} \wedge \dots (p_1 \wedge g_0)\right)\right)\right)\right)\right).$$

We will construct several families $(A_n)_{n\in\mathbb{N}}$ of adder circuits with a linear size and a very good depth. For this, we additionally use the following abbreviatory notation.

Notation. Given a circuit C, we write $d(C) := \operatorname{depth}(C)$ and $s(C) := \operatorname{size}(C)$.

Our core idea for adder optimization is to apply our And-Or path optimization algorithm from Chapter 3. Given $m \in \mathbb{N}$, by Corollary 3.4.21, this algorithm constructs a circuit for an And-Or path with length m with a depth of at most $\log_2 m + \log_2 \log_2 m + 1.58$ and linear size $4.15m - 4$. This is the best depth bound for And-Or path circuits known so far, and it is only by a constant away from the asymptotic lower bound due to Commentz-Walter [Com79] shown in Corollary 2.6.8. As depth optimization for adders and And-Or paths are equivalent tasks as long as no other objectives are regarded, applying our And-Or path algorithm for computing each carry bit separately solves the Adder Optimization Problem optimally up to a constant. But this yields circuits with a quadratic size, which is undesirable. Hence, the main goal of this chapter is to apply our And-Or path optimization algorithm in a careful way such that the total size remains linear.

To this end, we proceed in several steps: In Section 8.1, we construct a family of adder circuits $\left(A_n^1\right)_{n\in\mathbb{N}}$ with $d(A_n^1) \leq \log_2 n + \log_2 \log_2 n + 3.58$ and sub-quadratic size $s(A_n^1) \leq 6.6 n \log_2 n$. In Section 8.2, we will see a general method for linearizing the size of adder circuits, increasing the depth only by an additive term of $\log_2 \log_2 \log_2 n + \text{const}$. In Section 8.3, this will be applied to linearize the adder family $\left(A_n^1\right)_{n\in\mathbb{N}}$, which leads to an adder family $\left(A_n^2\right)_{n\in\mathbb{N}}$ with

$$d(A_n^2) \leq \log_2 n + \log_2 \log_2 n + \log_2 \log_2 \log_2 n + 8.6$$

and $s(A_n^2) \leq 22.7n$. If we allow the additive constant in the depth bound to increase from 8.6 to 9.4, then, applying our linearization framework with slightly different

Algorithm 8.1: 2-part adder construction framework

```
Input: $n \in \mathbb{N}$, $n \geq 2$, and $n$ input pairs $p_0, g_0, \dots, p_{n-1}, g_{n-1}$, adder circuits
       $(A_k)_{k<n}$, AND-prefix circuits $(S_k)_{k<n}$, AND-OR path circuits
       $(AOP_k)_{k<n}$.
Output: An adder circuit $C_n$ on $p_0, g_0, \dots, p_{n-1}, g_{n-1}$.
1  $k_l \leftarrow \lfloor \frac{n}{2} \rfloor$, $k_r \leftarrow \lceil \frac{n}{2} \rceil$.
2  $P_r \leftarrow (p_0, g_0, \dots, p_{k_r-1}, g_{k_r-1})$, $P_l \leftarrow (p_{k_r}, g_{k_r}, \dots, p_{n-1}, g_{n-1})$.
3  Compute an adder circuit $A_{k_l}$ on $P_l$.
4  Compute an adder circuit $A_{k_r}$ on $P_r$.
5  Compute an AND-prefix circuit $S_{k_l}$ on the inputs $p_i$ with $i > k_r$ of $P_{k_l}$.
6  Compute an AND-OR path circuit $AOP_{k_r}$ on the inputs of $P_{k_r}$.
7  for $i \leftarrow 1$ to $k_r$ do
8      Let $\text{out}_i(C_n) := \text{out}_i(A_{k_r})$.
9  for $i \leftarrow 1$ to $k_l$ do
10     Let $\text{out}_{k_r+i}(C_n) := \text{out}_i(A_{k_l}) \vee \Big(\text{out}_i(S_{k_l}) \wedge AOP_{k_r}\Big)$.
11 return $C_n$.
```

parameters, we can construct adder circuits A_n^3 with $s(A_n^3) \leq 17.6n$. The running time needed to construct A_n^2 is $\mathcal{O}(n \log_2 n)$, while it is $\mathcal{O}(n \log_2 \log_2 n)$ for A_n^3.

Hence, the depth of both A_n^2 and A_n^3 is by an additive term of $\log_2 \log_2 \log_2 n +$ const away from the lower bound on depth by Commentz-Walter [Com79] shown in Corollary 2.6.8. This has significantly improved the gap to the lower bound in comparison with the previously best depth of

$$\log_2 n + 8\left\lceil \sqrt{\log_2 n} \right\rceil + 6\left\lceil \log_2 \left\lceil \sqrt{\log_2 n} \right\rceil \right\rceil + 2$$

achieved by the adder circuits of Held and Spirkl [HS17a] (here, n needs to be a power of 2), for which the gap is in the order of $\mathcal{O}\left(\sqrt{\log_2 n}\right)$. The size of the circuits by Held and Spirkl is at most $13.5n$, and even at most $9.5n$ if $n \geq 4096$. As for their analysis, Held and Spirkl assume that n is a power of two, for arbitrary n, their depth bound increases by a constant, and their size bound increases up to a factor of 2. Thus, there are arbitrarily large instances where our circuits A_n^3 have a better size than the circuits by Held and Spirkl [HS17a].

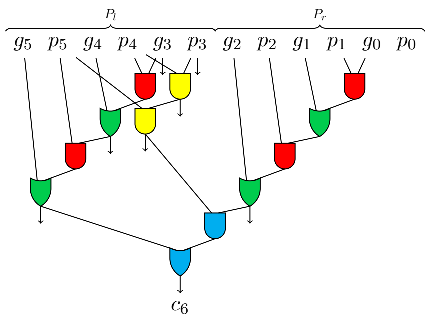
8.1 Fast Adder Circuits with Sub-Quadratic Size

Our idea for constructing fast adder circuits with a sub-quadratic size is to apply the AND-OR path optimization algorithm from Chapter 3 only for the computation of some carry bits. Hence, we use the generic procedure described in Algorithm 8.1 as a core routine for constructing our first family of adders.

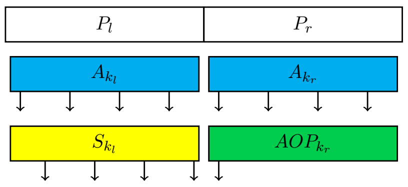
Algorithm 8.1 works as follows: We partition the input pairs into two parts $P_r = (p_0, g_0, \dots, p_{k_r-1}, g_{k_r-1})$ and $P_l = (p_{k_r}, g_{k_r}, \dots, p_{n-1}, g_{n-1})$ of roughly equal sizes k_l and k_r as defined in line 1. Note that we have

$$k_l + k_r = \left\lfloor \frac{n}{2} \right\rfloor + \left\lceil \frac{n}{2} \right\rceil = \begin{cases} n & n \text{ even} \\ \frac{n-1}{2} + \frac{n+1}{2}, & n \text{ odd} \end{cases} = n\,,$$

so this is indeed a correct partition of all input pairs. On the parts P_r and P_l, we construct an adder circuit A_{k_l} and A_{k_l}, respectively. Hence, for the part P_r, we have already computed all the carry bits for C_n in A_{k_r}. Additionally, we construct an And-Or path circuit AOP_{k_r} circuit on part P_r and an And-prefix circuit S_{k_r} on P_r. Using these two circuits and A_{k_l}, for the part P_l, the carry bits can be computed by applying the alternating split presented in Lemma 2.6.13, see also Figure 8.1(a). The carry bit c_{k_r} can be read off from AOP_{k_r}. Thus, Algorithm 8.1 correctly computes all the carry bits and hence an adder circuit on n input pairs.


P_l
P_r
g_5 p_5 g_4 p_4 g_3 p_3 g_2 p_2 g_1 p_1 g_0 p_0
c_6

(a) Computing the carry bit c_6 for 6 input pairs as in line 10 of Algorithm 8.1 via the alternating split from Lemma 2.6.13. On P_l, we indicate an adder circuit and an And-prefix circuit, and on P_r an And-Or path circuit by trivial implementations.



(b) All components constructed in Algorithm 8.1. On P_l, we see the adder circuit A_{k_l} in blue and the And-prefix circuit S_{k_l} in yellow; and on P_r, we see the adder circuit A_{k_r} in blue and the And-Or path circuit AOP_{k_r} in green.

Figure 8.1: Illustration of Algorithm 8.1.

In Figure 8.1(b), we depict all circuits used to construct the adder circuit in Algorithm 8.1. The following lemma estimates the depth and size of this circuit.

Lemma 8.1.1. *Given $n \in \mathbb{N}$ with $n \geq 2$, adder circuits $(A_k)_{k<n}$,* And*-prefix circuits $(S_k)_{k<n}$, and* And-Or *path circuits $(AOP_k)_{k<n}$, Algorithm 8.1 computes an adder circuit C_n with*

$$d(C_n) \leq \max\Big\{d\big(A_{k_r}\big), d\big(A_{k_l}\big) + 1, d\big(AOP_{k_r}\big) + 2, d\big(S_{k_l}\big) + 2\Big\}$$

and

$$s(C_n) \leq s\big(A_{k_r}\big) + s\big(A_{k_l}\big) + s\big(AOP_{k_r}\big) + s\big(S_{k_l}\big) + 2k_l\,.$$

Proof. The size of C_n arises from adding up the sizes of all sub-circuits, increased by $2k_l$ since for each $i \in \{k_r + 1, \dots, n\}$, the ith carry bit is computed using two additional gates in line 10 (in Figure 8.1(a), for c_6, these are the two blue gates).

For the depth estimation, we consider two different cases. If $i \leq k_r$, the depth of the ith carry bit is simply the depth of A_{k_r}, see line 8. If $i > k_r$, the ith carry bit is computed in line 10 with a depth of at most

$$\max\Big\{d\big(A_{k_l}\big) + 1, d\big(AOP_{k_r}\big) + 2, d\big(S_{k_l}\big) + 2\Big\}.$$

As the depth of the adder circuit C_n is the maximum depth of any of its carry bits, the depth bound follows. □

Note that in Algorithm 8.1, we actually compute the carry bit $\text{out}_{k_r}(C_n)$ twice – once by the And-Or path, once by the last output of A_{k_r}. We do this as it does not harm the overall analysis and simplifies notations.

We now derive an adder family $\big(A_n^1\big)_{n \in \mathbb{N}}$ with a good depth and size in the order of $\mathcal{O}(n \log_2 n)$. For small n, we construct existing adder circuits. For large n, we use Algorithm 8.1 in a recursive fashion: We apply Algorithm 8.1 with the AND-OR path circuit from Corollary 3.4.21 as AOP_{k_r} and the circuit $S_{k_l}^f$ by Ladner and Fischer [LF80] with $f = 3$ as S_{k_l}, see also Theorem 2.6.24. The adder circuits A_{k_l} and A_{k_r} are computed recursively using Algorithm 8.1. In order to bound the depth and size of the arising circuits, we need two numerical inequalities.

Lemma 8.1.2. *For $3 \leq n \leq 20$, the following statements are fulfilled.*

(i) We have $0.441 \log_2 n \leq \log_2 \log_2 n + 0.906$.

(ii) We have $1.5\big(n^2 - n\big) \leq 6.6 n \log_2 n$.

Proof. We prove the first statement via a case distinction: For $3 \leq n \leq 11$, we have

$$0.441 \log_2 n \overset{n \leq 11}{<} 1.53 < 1.57 \overset{n \geq 3}{<} \log_2 \log_2 n + 0.906\,;$$

and for $12 \leq n \leq 20$, we have

$$0.441 \log_2 n \overset{n \leq 20}{<} 2 < 2.7 \overset{n \geq 12}{<} \log_2 \log_2 n + 0.906\,.$$

To prove the second statement, we need that by Lemma 3.2.2, the function $x \mapsto \frac{2^x}{x}$ is monotonely increasing for $x \geq 2$. Hence, the function $x \mapsto \frac{\log_2 x}{x}$ is monotonely decreasing for $x \geq 4$. Consequently, for $n \geq 4$, we have

$$\begin{aligned}
6.6 \log_2 n - 1.5(n-1) \quad &= \quad n\left(\frac{6.6 \log_2 n + 1.5}{n} - 1.5\right) \\
&\overset{\substack{4 \leq n \leq 20,\\ \text{Lem. 3.2.2}}}{\geq} \quad n\left(\frac{6.6 \log_2 20 + 1.5}{20} - 1.5\right) \\
&> \quad n(1.501 - 1.5) \\
&> \quad 0\,.
\end{aligned}$$

Multiplying with n yields the second statement for $n \geq 4$. For $n = 3$, we have

$$1.5\Big(n^2 - n\Big) = 9 < 31 < 6.6 n \log_2 n\,,$$

so the second statement is fulfilled for all $3 \leq n \leq 20$. □

The following theorem describes our concrete strategy for computing the adder family $\big(A_n^1\big)_{n \in \mathbb{N}}$ and analyzes the resulting depth and size.

Theorem 8.1.3. *Let $n \in \mathbb{N}$ with $n \geq 3$ and $c := 6.6$ be given. We can construct an adder circuit A_n^1 on n input pairs with depth*

$$d(A_n^1) \leq \log_2 n + \log_2 \log_2 n + 3.58$$

and size

$$s(A_n^1) \leq c n \log_2 n$$

in running time $\mathcal{O}\Big(n \log_2^2 n\Big)$.

Proof. First, by induction on n, we will see the depth and size bounds.

Case 1: Assume that $3 \leq n \leq 20$.

We apply the AND-OR path optimization algorithm by Held and Spirkl [HS17b] to compute each carry bit separately as described in Corollary 2.6.29. The arising circuit A_n^1 has depth

$$d(A_n^1) \overset{\text{Cor. 2.6.29}}{\leq} 1.441 \log_2 n + 2.674 \overset{\substack{n \leq 20, \\ \text{Lem. 8.1.2,}(i)}}{\leq} \log_2 n + \log_2 \log_2 n + 3.58$$

and size

$$s(A_n^1) \overset{\text{Cor. 2.6.29}}{\leq} 1.5(n^2 - n) \overset{\substack{n \leq 20, c=6.6, \\ \text{Lem. 8.1.2,}(ii)}}{\leq} cn \log_2 n\,.$$

Case 2: Assume that $n \geq 21$.

We may assume inductively that for all $i < n$, an adder A_i^1 with the stated depth and size can be constructed. Hence, we may apply Algorithm 8.1 using the following sub-circuits:

- We use $A_{k_l} := A_{k_l}^1$ and $A_{k_r} := A_{k_r}^1$. By induction hypothesis, we have

$$\begin{aligned} d\big(A_{k_l}\big) &\overset{\text{(IH)}}{\leq} \log_2(k_l) + \log_2 \log_2(k_l) + 3.58 \\ &\overset{k_l \leq \frac{n}{2}}{\leq} \log_2 n + \log_2 \log_2 n + 2.58\,, && (8.1) \\ d\big(A_{k_r}\big) &\overset{\text{(IH)}}{\leq} \log_2(k_r) + \log_2 \log_2(k_r) + 3.58 \\ &\overset{k_r \leq n}{\leq} \log_2 n + \log_2 \log_2 n + 3.58\,, && (8.2) \\ s\big(A_{k_l}\big) &\overset{\text{(IH)}}{\leq} ck_l \log_2(k_l)\,, && (8.3) \\ s\big(A_{k_r}\big) &\overset{\text{(IH)}}{\leq} ck_r \log_2(k_r)\,. && (8.4) \end{aligned}$$

- For the computation of the AND-prefix circuit S_{k_l} on the k_l input pairs of P_l, we use the Ladner-Fischer circuit $S_{k_l}^3$, see Theorem 2.6.24 and [LF80]. Note that the choice $f = 3$ fulfills the requirement $f \overset{n \geq 9}{\leq} \lceil \log_2 k_l \rceil$. Then, we have

$$d(S_{k_l}) \leq \lceil \log_2 k_l \rceil + 3 = \left\lceil \log_2 \left\lfloor \frac{n}{2} \right\rfloor \right\rceil + 3 \leq \log_2 n + 3 \qquad (8.5)$$

  and

$$s(S_{k_l}) \leq 2\left(1 + \frac{1}{2^3}\right)k_l = 2.25k_l\,. \qquad (8.6)$$

- As AND-OR path circuit AOP_{k_r}, we use the circuit from Corollary 3.4.21. Since $k_r \leq \frac{n+1}{2}$ and AOP_{k_r} has $2k_r - 1 \leq n$ inputs, by Corollary 3.4.21, we have

$$d(AOP_{k_r}) \leq \log_2 n + \log_2 \log_2 n + 1.58 \qquad (8.7)$$

  and

$$s(AOP_{k_r}) \leq 4.15n - 4\,. \qquad (8.8)$$

For the depth of A_n^1, these observations together with Lemma 8.1.1 imply

$$\begin{aligned} d(A_n^1) &\overset{\text{Lem. 8.1.1}}{\leq} \max\Big\{d\big(A_{k_r}\big), d\big(A_{k_l}\big)+1, d\big(AOP_{k_r}\big)+2, d\big(S_{k_l}\big)+2\Big\} \\ &\overset{\substack{(8.1),(8.2),\\(8.7),(8.5)}}{\leq} \max\{\log_2 n + \log_2\log_2 n + 3.58, \log_2 n + 5\} \\ &\overset{n\geq 7}{\leq} \log_2 n + \log_2\log_2 n + 3.58\,. \end{aligned}$$

It remains to compute the size of A_n^1. As $n \geq 21$, we have

$$\begin{aligned} \log_2 k_r &\overset{k_r=\lceil \frac{n}{2}\rceil}{\leq} \log_2\left(\frac{n+1}{2}\right) \\ &= \log_2\left(n\cdot\frac{n+1}{2n}\right) \\ &= \log_2 n - 1 + \log_2\left(1+\frac{1}{n}\right) \\ &\overset{n\geq 21}{\leq} \log_2 n - 0.932\,. \end{aligned} \tag{8.9}$$

Moreover, for $0 \leq \alpha \leq 1$, we have

$$\alpha k_r + k_l = \begin{cases} (\alpha+1)\frac{n}{2} & \text{if } n \text{ even} \\ \alpha\frac{n+1}{2}+\frac{n-1}{2} & \text{if } n \text{ odd} \end{cases} \overset{\alpha\leq 1}{\geq} n\frac{\alpha+1}{2}+\frac{\alpha-1}{2}\,. \tag{8.10}$$

Based on these two inequalities, we can bound the total size of the recursively computed adder circuits by

$$\begin{aligned} s(A_{k_r})+s(A_{k_l}) &\overset{(8.4),(8.3)}{\leq} ck_r\log_2(k_r)+ck_l\log_2(k_l) \\ &\overset{(8.9),k_l\leq\frac{n}{2}}{\leq} c\big(k_r(\log_2 n-0.932)+k_l(\log_2 n-1)\big) \\ &\overset{k_r+k_l=n}{=} c(n\log_2 n - 0.932k_r - k_l) \\ &\overset{(8.10)}{\leq} c\left(n\log_2 n - n\frac{0.932+1}{2}-\frac{0.932-1}{2}\right) \\ &= c\Big(n\Big(\log_2 n - 0.966\Big)+0.034\Big)\,. \end{aligned} \tag{8.11}$$

In total, the size of A_n^1 is hence at most

$$\begin{aligned} s(A_n^1) &\overset{\text{Thm. 8.1.1}}{\leq} s(A_{k_r})+s(A_{k_l})+s(AOP_{k_r})+s(S_{k_l})+2k_l \\ &\overset{(8.11),(8.8),(8.6)}{\leq} c\Big(n\Big(\log_2 n-0.966\Big)+0.034\Big)+4.15n+2.25k_l+2k_l \\ &\overset{k_l\leq\frac{n}{2}}{\leq} cn\log_2 n - 0.966cn + 0.034c + 6.275n \\ &= cn\left(\log_2 n - 0.966 + \frac{0.034}{n}+\frac{6.275}{c}\right) \\ &\overset{\substack{n\geq 21,\\c\geq 6.6}}{\leq} cn\Bigg(\log_2 n - 0.966 + 0.002 + 0.951\Bigg) \\ &< cn\log_2 n\,. \end{aligned}$$

Finally, for proving the running time bound, it suffices to consider the case $n \geq 21$. We show that there is a constant α such that we can compute our circuit in $\alpha n \log^2 n$ elementary steps.

In each call of Algorithm 8.1, we apply two recursive calls to instances with size of at most $\frac{n+1}{2}$ each, so we may assume inductively that each of them can be performed with at most $\alpha \frac{n+1}{2} \log^2\left(\frac{n+1}{2}\right)$ steps. By Corollary 3.4.21, there is a constant β such that the And-Or path AOP_{k_r} with at most n inputs can be computed with $\beta n \log_2 n$ steps. The rest of one call of Algorithm 8.1 (including the computation of $S_{k_l} = S^3_{k_l}$, see Proposition 2.6.25) takes linear time, so assume that we need γn steps for this.

For $n \geq 21$, we have

$$\begin{aligned} \alpha n \log^2\left(\frac{n+1}{2}\right) &\overset{n\geq 21}{\leq} \alpha n \log_2^2\left(\frac{n}{1.9}\right) \\ &< \alpha n (\log_2 n - 0.96)^2 \\ &= \alpha n \log_2^2 n - 1.92 \alpha n \log_2 n + 0.9216 \alpha n\,, \end{aligned} \tag{8.12}$$

so in total, the number of steps for computing A^1_n can be bounded by

$$\begin{aligned} & 2\alpha \frac{n+1}{2} \log^2\left(\frac{n+1}{2}\right) + \beta n \log_2 n + \gamma n \\ = \; & \alpha n \log^2\left(\frac{n+1}{2}\right) + \alpha \log^2\left(\frac{n+1}{2}\right) + \beta n \log_2 n + \gamma n \\ \overset{(8.12)}{\leq} \; & \alpha n \log_2^2 n - 1.92\alpha n \log_2 n + 0.9216 \alpha n + \alpha \log^2\left(\frac{n+1}{2}\right) + \beta n \log_2 n + \gamma n\,. \end{aligned}$$

This is at most $\alpha n \log^2 n$ if α is chosen sufficiently large compared to β and γ. $\square$

8.2 An Adder Linearization Framework

In this section, we develop a framework that linearizes a given adder family $(B_n)_{n\in\mathbb{N}}$ with sub-quadratic size in a way that depth does not increase too much. The idea of this linearization goes back to Ofman [Ofm62] and Khrapchenko [Khr67], but we perform some crucial changes in order to obtain a best possible depth. See Remark 8.2.7 for a comparison of the two linearization frameworks.

Algorithm 8.2 is a method for solving the Adder Optimization Problem on n input pairs that is an extension of Algorithm 8.1. It depends on oracles solving the Adder Optimization Problem, the And-Or path Circuit Depth Optimization Problem and the Parallel And-Prefix Problem. In Theorem 8.2.4, we present our linearization framework which uses this algorithm.

Algorithm 8.2 constructs a circuit C_n as follows: Given $k \in \mathbb{N}_{>0}$ and $l = \left\lceil \frac{n}{k} \right\rceil$, in lines 1 to 6, we partition the inputs into l consecutive parts $P^{(0)}, \ldots, P^{(l-1)}$, where each group has $n_j \leq k$ input pairs. On each part $P^{(j)}$, we compute an adder circuit $A^{(j)}_{n_j}$. In part $P^{(0)}$, we can directly read off the carry bits of C_n from $A^{(0)}_{n_j}$ in line 9. For computing the carry bits of C_n in part $P^{(j)}$ for $j > 0$, as in Algorithm 8.1, we use the alternating split from Lemma 2.6.13 in line 16. For this, we need an adder circuit $A^{(j)}_{n_j}$ and an And-prefix circuit $S^{(j)}_{n_j}$ on $P^{(j)}$, and an And-Or path circuit $AOP^{(j)}_{N_j}$ on the input parts $P^{(j-1)}, \ldots, P^{(0)}$. These are constructed in lines 11 to 13.

Algorithm 8.2: l-part adder construction framework

```
Input: $n \in \mathbb{N}$, $n \geq 2$, $n$ input pairs $p_0, g_0, \dots, p_{n-1}, g_{n-1}$, a family of adder
       circuits $(A_k)_{k \in \mathbb{N}}$, a family of AND-prefix circuits $(S_k)_{k \in \mathbb{N}}$, a family
       of AND-OR path circuits $(AOP_k)_{k \in \mathbb{N}}$.
Output: An adder circuit $C_n$ on $p_0, g_0, \dots, p_{n-1}, g_{n-1}$.
1  Choose $k \in \mathbb{N}_{>0}$ and $l := \lceil n/k \rceil$.
2  for $j \leftarrow 0$ to $l-2$ do
3      $P^{(j)} := (p_{jk}, g_{jk}, \dots, p_{(j+1)k-1}, g_{(j+1)k-1})$.
4  $P^{(l-1)} := (p_{(l-1)k}, g_{(l-1)k}, \dots, p_{n-1}, g_{n-1})$.
5  for $j \leftarrow 0$ to $l-1$ do
6      $n_j \leftarrow |P^{(j)}|$, $N_j \leftarrow n_0 + \ldots + n_{j-1}$.
7  Construct an adder circuit $A^{(0)}_{n_0}$ on $P^{(0)}$.
8  for $i \leftarrow 1$ to $n_0$ do
9      Let $\text{out}_i(C_n) := \text{out}_i\Big(A^{(0)}_{n_0}\Big)$.
10 for $j \leftarrow 1$ to $l-1$ do
11     Construct an adder circuit $A^{(j)}_{n_j}$ on $P^{(j)}$.
12     Construct an AND-prefix circuit $S^{(j)}_{n_j}$ on $P^{(j)}$.
13     Construct an AND-OR path circuit $AOP^{(j)}_{N_j}$ on the $N_j$ input pairs in
        $P^{(j-1)}, \dots, P^{(0)}$.
14     Let $\text{out}_{N_j}(C_n) := AOP^{(j)}$.
15     for $i \leftarrow 1$ to $n_j$ do
16         Let $\text{out}_{N_j+i}(C_n) := \text{out}_i\Big(A^{(j)}_{n_j}\Big) \lor \Big(\text{out}_i\Big(S^{(j)}_{n_j}\Big) \land AOP^{(j)}_{N_j}\Big)$.
17 return $C_n$.
```

Hence, all carry bits are computed correctly and C_n is an adder circuit on n input pairs. Figure 8.2 illustrates the circuits computed for part $P^{(j)}$.

Note that Algorithm 8.1 is a special case of Algorithm 8.2 where $k = \lceil \frac{n}{2} \rceil$.

It is easy to read off the depth and size of C_n from the construction:

Observation 8.2.1. Given $n \in \mathbb{N}$, $n \geq 2$, the circuit C_n computed by Algorithm 8.2 has depth

$$\begin{aligned} d(C_n) &\leq \max\left\{ d(A_{n_0}), \max_{j \in \{1,\dots,l-1\}} \left\{ d(A_{n_j}) + 1, \max\left\{ d(S_{n_j}), d(AOP_{N_j}) \right\} + 2 \right\} \right\} \\ &\overset{n_j \leq k}{\leq} \max\left\{ d(A_k) + 1, d(S_k) + 2, \max_{j \in \{1,\dots,l-1\}} \left\{ d(AOP_{N_j}) + 2 \right\} \right\} \end{aligned}$$

and size

$$\begin{aligned} s(C_n) &\leq s\big(A_{n_0}\big) + \sum_{j=1}^{l-1} \Big(s(A_{n_j}) + s(S_{n_j}) + s(AOP_{N_j}) + 2(n_j - 1) \Big) \\ &< s(A_{n_0}) + \sum_{j=1}^{l-1} \Big(s(A_{n_j}) + s(S_{n_j}) + s(AOP_{N_j}) \Big) + 2n\,. \end{aligned}$$

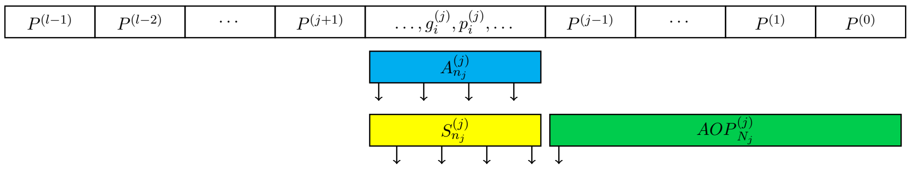


Figure 8.2: Illustration of the circuits constructed for part $P^{(j)}$ in lines 11 to 13 of Algorithm 8.2.

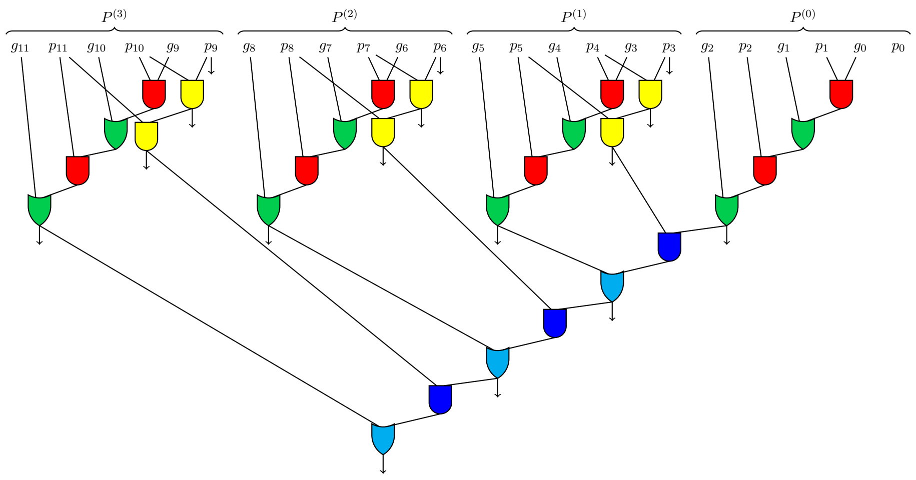

$P^{(3)}$
$P^{(2)}$
$P^{(1)}$
$P^{(0)}$
g_{11} p_{11} g_{10} p_{10} g_9 p_9
g_8 p_8 g_7 p_7 g_6 p_6
g_5 p_5 g_4 p_4 g_3 p_3
g_2 p_2 g_1 p_1 g_0 p_0

Figure 8.3: Circuit arising from applying Corollary 8.2.2 to $l = 4$ input parts with $k = 3$ input pairs each.

Algorithm 8.2 will be the main ingredient of our linearization framework. However, we will not linearize the size of the circuit A_k, but of an adder circuit B_l with l input pairs which we will introduce now. For this, we compute the And-Or paths AOP_{N_j} for $j = 1, \ldots, l-1$ in line 13 in a special way. Here, given $j \in \{0, \ldots, l-1\}$, we write $a(P^{(j)}) := \operatorname{sym}\Big(\big(p_{jk}, p_{jk+1}, \ldots, p_{jk+n_j-1}\big)\Big)$ for the And function on all "p-inputs" of part $P^{(j)}$, and $h^*(P^{(j)}) := g^*(g_{jk+n_j-1}, p_{jk+n_j-1}, \ldots, g_{jk+1}, p_{jk+1}, g_{jk})$ for the And-Or path on all inputs of $P^{(j)}$ but p_{jk}.

Corollary 8.2.2. *Let n pairs of Boolean input variables $g_0, p_1, g_1 \ldots, p_{n-1}, g_{n-1}$ and a partition of the input pairs into l parts $P^{(l-1)}, \ldots, P_0$ be given. We have*

$$\begin{aligned} &g^*(g_{n-1}, p_{n-1}, \ldots, g_1, p_1, g_0) \\ &\quad = g^*\Big(h^*(P^{(l-1)}), a(P^{(l-1)}), \ldots, h^*\Big(P^{(1)}), a(P^{(1)}), h^*(P^{(0)})\Big)\Big). \end{aligned}$$

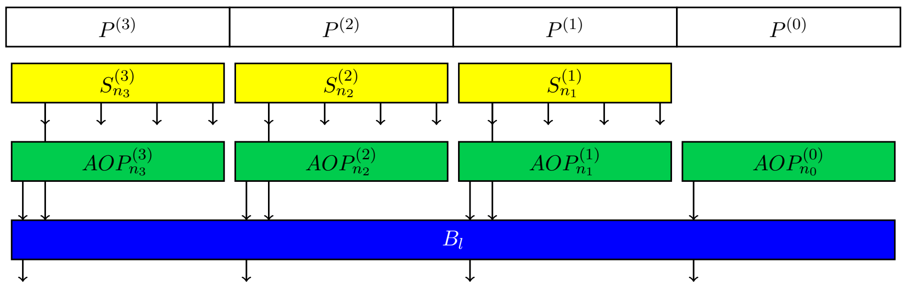


Figure 8.4: Illustration of the construction of the AND-OR paths $AOP^{(j)}_{N_j}$ for each $j \in \{0, \dots, l-1\}$ as in Lemma 8.2.3.

Proof. Applying Lemma 2.6.13 iteratively yields

$$
\begin{aligned}
&g^*(g_{n-1}, p_{n-1}, \dots, g_0)\\
&= h^*(P^{(l-1)}) \lor \left(a(P^{(l-1)}) \land g^*\left(P^{(l-2)} \mathbin{+\!\!+} \dots \mathbin{+\!\!+} P^{(0)}\right)\right)\\
&= h^*(P^{(l-1)}) \lor \left(a(P^{(l-1)}) \land \left(h^*(P^{(l-2)}) \lor \left(a(P^{(l-2)}) \land (\dots \lor (a(P_1) \land h^*(P_0)))\right)\right)\right)\\
&= g^*\left(h^*(P^{(l-1)}), a(P^{(l-1)}), \dots, h^*\left(P^{(1)})\right), a(P^{(1)})), h^*(P^{(0)})\right)\right). \qquad \square
\end{aligned}
$$

Figure 8.3 illustrates this corollary on $l = 4$ input parts: In yellow, we show the AND trees on every part $P^{(j)}$, in red and green the AND-OR paths on every part $P^{(j)}$, and in dark and light blue the AND-OR path that has the outputs of the former mentioned circuits as inputs. In order to compute all the AND-OR paths AOP_{N_j} for $j = 1, \dots, l-1$, we need the output of the blue AND-OR path after each OR gate. Hence, the blue part can be realized by an adder circuit as in the following lemma.

Lemma 8.2.3. *In addition to the input of Algorithm 8.2, let a family $(B_k)_{k\in\mathbb{N}}$ of adder circuits be given. For each $j \in \{0, \dots, l-1\}$, compute an* AND-OR *path circuit AOP_{n_j} on input part $P^{(j)}$. Compute an adder circuit B_l on l input pairs*

$$AOP^{(0)}_{n_0}, \mathrm{out}_{n_1}\left(S^{(1)}_{n_1}\right), AOP^{(1)}_{n_1}, \dots, \mathrm{out}_{n_{l-1}}\left(S^{(l-1)}_{n_{l-1}}\right), AOP^{(l-1)}_{n_{l-1}}.$$

For $j \in \{1, \dots, l-1\}$, we have $AOP^{(j)}_{N_j} = \mathrm{out}_j(B_l)$.

Proof. This is a direct consequence of Corollary 8.2.2. □

Our linearization framework is Algorithm 8.2 with this lemma applied for the computation of the AND-OR paths $AOP^{(j)}_{N_j}$ in line 13 of Algorithm 8.2. Figure 8.5 shows all the circuits used in the linearization framework. These are the circuits used in the adder construction framework from Figure 8.2, where the AND-OR paths $AOP^{(j)}_{N_j}$ for each $j \in \{0, \dots, l-1\}$ are computed using Lemma 8.2.3 as depicted in Figures 8.3 and 8.4. Here, we need another adder B_l and AND-OR paths $AOP^{(j)}_{n_j}$. After analyzing the depth and size of the resulting family of adders, we shall explain why this is actually a linearization framework for the adder family $(B_l)_{l\in\mathbb{N}}$.

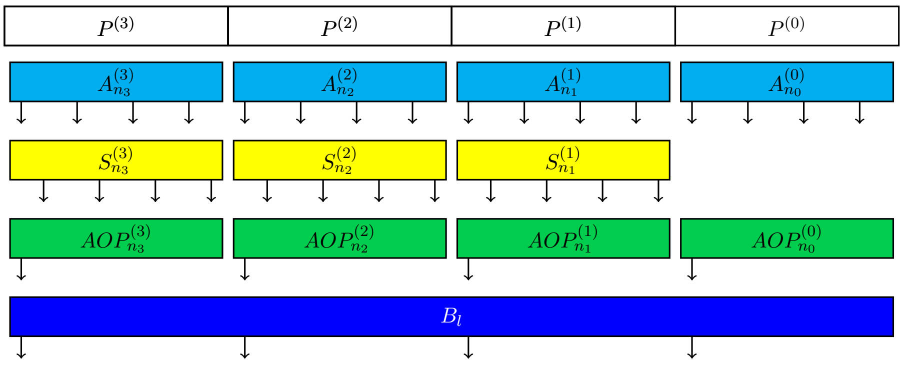


Figure 8.5: Illustration of the adder linearization framework from Theorem 8.2.4.

Theorem 8.2.4. *Let $n \in \mathbb{N}$ with $n \geq 2$, two families of adder circuits $(A_k)_{k\in\mathbb{N}}$ and $(B_l)_{l\in\mathbb{N}}$, a family of* And*-prefix circuits $(S_k)_{k\in\mathbb{N}}$, and a family of* And-Or *path circuits $(AOP_k)_{k\in\mathbb{N}}$ be given. Using Lemma 8.2.3 for the computation of the* And-Or *paths in line 13, Algorithm 8.2 computes an adder circuit C_n on n input pairs with depth*

$$d(C_n) \leq \max\Big\{d(A_k) + 1, d(B_l) + \max\{d(AOP_k), d(S_k)\} + 2\Big\}$$

and size

$$s(C_n) \leq s(A_{n_0}) + \sum_{j=1}^{l-1}\Big(s(A_{n_j}) + s(AOP_{n_j}) + s(S_{n_j})\Big) + s(B_l) + 2n.$$

Proof. By construction, the depth of AOP_{N_j} can be bounded by

$$\begin{aligned} d(AOP_{N_j}) &\overset{\text{Lem. 8.2.3}}{\leq} d(B_l) + \max\Big\{d(AOP_{n_0}), \max_{j=1,\dots,l-1}\Big\{d(AOP_{n_j}), d(S_{n_j})\Big\}\Big\} \\ &\overset{n_j \leq k}{\leq} d(B_l) + \max\big\{d(AOP_k), d(S_k)\big\}. \end{aligned} \tag{8.13}$$

Hence, we obtain a total depth of

$$\begin{aligned} d(C_n) &\overset{\text{Thm. 8.2.1}}{\leq} \max\Big\{d(A_k) + 1, d(S_k) + 2, \max_{j\in\{1,\dots,l-1\}}\Big\{d\Big(AOP_{N_j}\Big) + 2\Big\}\Big\} \\ &\overset{(8.13)}{\leq} \max\Big\{d(A_k) + 1, d(S_k) + 2, d(B_l) + \max\big\{d(AOP_k), d(S_k)\big\} + 2\Big\} \\ &= \max\Big\{d(A_k) + 1, d(B_l) + \max\big\{d(AOP_k), d(S_k)\big\} + 2\Big\}. \end{aligned}$$

The size bound follows directly from Observation 8.2.1 and Lemma 8.2.3. □

The preceding theorem gives a construction method for a family of adder circuits $(C_n)_{n\in\mathbb{N}}$ based on other families of adder circuits $(B_l)_{l\in\mathbb{N}}$ and $(A_k)_{k\in\mathbb{N}}$, And-Or path circuits and And-prefix circuits. We will see in Theorem 8.2.6 that this is actually a linearization of the family $(B_l)_{l\in\mathbb{N}}$ once the sizes of the And-Or path circuits, And-prefix circuits, and $(A_k)_{k\in\mathbb{N}}$ are linear. For this, we will use the adder circuits from the following proposition as family $(A_k)_{k\in\mathbb{N}}$.

Proposition 8.2.5. *For each $n \in \mathbb{N}$, there is an adder circuit A_n with $s(A_n) \leq 3.5n$ and $d(A_n) \leq n + 2$.*

Proof. For $n \leq 1$, we can construct an adder circuit with depth and size 0.

For $n \geq 2$, we apply Algorithm 8.1 to the following circuit families:

- We use the ripple-carry adder (see Observation 2.4.6) to compute the adder circuits A_{k_l} and A_{k_r}. From A_{k_r}, we can also read off the AND-OR path AOP_{k_r}.
- For S_{k_l}, we use an AND-path $S_{k_l} = p_{n-1} \wedge (p_{n-2} \wedge (\ldots \wedge (p_{k_r+1} \wedge p_{k_r+1})))$ as AND-prefix circuit on P_l.

Denote the resulting circuit by A_n. The depth of A_n is at most

$$\begin{aligned} d(A_n) &\overset{\text{Lem. 8.1.1}}{\leq} \max\{d(A_{k_l}) + 1, d(A_{k_r}) + 2, d(S_{k_l}) + 2\} \\ &\overset{\text{Obs. 2.4.6}}{=} \max\{2k_l - 1, 2k_r, k_l\} \\ &= 2k_r \\ &\leq n + 2\,. \end{aligned}$$

As A_{k_r} and AOP_{k_r} have identical gates, the size of A_n is

$$\begin{aligned} s(A_n) &\overset{\text{Lem. 8.1.1}}{\leq} s(A_{k_r}) + s(A_{k_l}) + s(S_{k_l}) + 2k_l \\ &\overset{\text{Obs. 2.4.6}}{\leq} 2k_r - 2 + 2k_l - 2 + k_l - 2 + 2k_l \\ &< 2k_r + 5k_l \\ &\leq 3.5n\,. \end{aligned}$$

□

Theorem 8.2.6. *Let a family of adder circuits $(B_l)_{l\in\mathbb{N}}$ with depth $d(B_l) \leq \log_2 l + \delta(l)$ and size $s(B_l) \leq l\sigma(l)$ be given, where $\sigma, \delta\colon \mathbb{N} \to \mathbb{R}_{\geq 0}$ are monotonely increasing functions. Then, for any $n \in \mathbb{N}$, there is a linear-size adder circuit C_n with*

$$d(C_n) \leq \max\big\{\,\sigma(n), d(B_n) + \log_2\log_2(\sigma(n))\,\big\} + \textit{const}\,.$$

Proof. We apply the linearization from Theorem 8.2.4 to the adder family $(B_l)_{l\in\mathbb{N}}$, using the following other circuits: For the adder family $(A_k)_{k\in\mathbb{N}}$, we use the circuits from Proposition 8.2.5; for the AND-prefix circuit family $(S_k)_{k\in\mathbb{N}}$, we use the Ladner-Fischer circuits $\big(S_k^0\big)_{k\in\mathbb{N}}$ from Theorem 2.6.24 with $f = 0$; and for the AND-OR paths AOP_k, we use the circuit from Corollary 3.4.21 (with $m \leq 2k$ inputs since we have k input pairs). Thus, we have

$$\begin{aligned} d(A_k) &\leq k + 2\,, & s(A_k) &\leq 3.5k\,, \\ d(AOP_k) &\leq \log_2(2k) + \log_2\log_2(2k) + 1.58\,, & s(AOP_k) &\leq 9.3k - 4\,, \\ d(S_k) &\leq \lceil \log_2 k \rceil\,, & s(S_k) &\leq 2\left(1 + \frac{1}{2^0}\right)k = 4k\,. \end{aligned}$$

We choose $k := \lceil\,\sigma(n)\,\rceil$, and $l := \lceil\,\frac{n}{k}\,\rceil$. This implies

$$s(B_l) \in \mathcal{O}\big(l\sigma(l)\big) = \mathcal{O}\left(\frac{n}{\sigma(n)}\sigma(n)\right) = \mathcal{O}(n)\,.$$

As also the sizes of A_k, AOP_k, and S_k are linear in k, we obtain

$$\begin{aligned}
&s(C_n)\\
\overset{\text{Thm. 8.2.4}}{\leq}\quad & s(A_{n_0}) + \sum_{j=1}^{l-1}\Big(s(A_{n_j}) + s(AOP_{n_j}) + s(S_{n_j})\Big) + s(B_l) + 2n\\
\in\quad & \mathcal{O}(n)\,.
\end{aligned}$$

For analyzing the depth of C_n, note that $d(AOP_k) \geq d(S_k)$. We conclude

$$\begin{aligned}
d(C_n) &\overset{\text{Thm. 8.2.4}}{\leq} \max\Big\{\, d(A_k)+1, d(B_l) + \max\big\{d(AOP_k), d(S_k)\big\} + 2\,\Big\}\\
&\leq \max\big\{\, k+3, \log_2 l + \delta(l) + \log_2(2k) + \log_2\log_2(2k) + 3.58\,\big\}\\
&= \max\big\{\, \sigma(n), \log_2 n - \log_2 k + \delta(l) + \log_2 k + \log_2\log_2 k\,\big\} + \text{const}\\
&\leq \max\big\{\, \sigma(n), d(B_n) + \log_2\log_2(\sigma(n))\,\big\} + \text{const}\,. \qquad \square
\end{aligned}$$

In particular, if the size of B_l is in the order of $\mathcal{O}(n\log_2 n)$, as for the adder from Theorem 8.1.3, we can linearize the adder family $(B_l)_{l\in\mathbb{N}}$ with a depth increase of only $\log_2\log_2\log_2 n$ (up to an additive constant). When the size of B_l is even smaller, the depth increase will also become smaller.

Remark 8.2.7. Our linearization framework presented in Theorems 8.2.4 and 8.2.6 is closely related to the linearization by Ofman [Ofm62] and Khrapchenko [Khr67], see Gashkov, Grinchuk, and Sergeev [GGS07] for a concise description. In Corollary 3.4.21, we present the first family of And-Or path circuits with asymptotically optimum depth of $\log_2 n + \log_2\log_2 n + \text{const}$ and, at the same time, a linear size. Hence, we use these circuits for construction of the And-Or paths AOP_{n_j} in Theorem 8.2.4.

Instead, Ofman [Ofm62] and Khrapchenko [Khr67] use last carry bit of the adder circuits A_k to compute AOP_{n_j}. This way, the depth of A_k is critical for the depth of C_n – very different to our approach, where in Theorem 8.2.6, we may use a circuit with a high depth of $k+3$. The best possible result for Ofman [Ofm62] and Khrapchenko [Khr67] is achieved by applying their linearization iteratively, which lead to a much higher depth increase in the order of $\mathcal{O}\Big(\sqrt[4]{\log_2 n}\Big)$ and linear-size adder circuits with a depth of $\log_2 n + \sqrt{2\log_2 n} + \mathcal{O}\Big(\sqrt[4]{\log_2 n}\Big)$.

Another linearization technique, which is also based on Khrapchenko [Khr67], is described concisely in Held and Spirkl [HS17a]. Given an adder family $(B_l)_{l\in\mathbb{N}}$ and $\tau \leq \log_2 n - 1$, this linearization constructs adder circuits with depth at most $d(B_{n/2^\tau}) + 2\tau$ and size at most $s(B_{n/2^\tau}) + 5n$. If we used this linearization with our family of adder circuits from Theorem 8.1.3 with a size of $\mathcal{O}(n\log_2 n)$ as B_l, we would need to choose $\tau = \log_2\log_2 n$. This would lead to a much larger depth increase in the order of $\mathcal{O}(\log_2\log_2 n)$.

Note that in Theorem 8.2.6, we omit the additive constant in the depth and the multiplicative constant in the size of the linearized adder C_n. In order to tune these constants, we will perform a more careful analysis in the next section. Moreover, we use slightly different sub-circuits than in Theorem 8.2.6.

8.3 Linear-Size Adder Circuits

Using the linearization framework presented in Theorem 8.2.4, we can now linearize the size of the adder circuits from Theorem 8.1.3. We discuss two results: First, in Theorem 8.3.6, we present a linear-size adder with the best depth we can obtain with our approach. Secondly, in Proposition 8.3.7, we show how to decrease the size of this adder substantially if the depth is allowed to increase by a small constant.

Again, in both cases, we will use other adder circuits when the number of inputs is small. For this, we need a technical lemma.

Lemma 8.3.1. *For $4 \leq n \leq 8192$, we have*

$$2\lceil \log_2 n \rceil \leq \log_2 n + \log_2 \log_2 n + \log_2 \log_2 \log_2 n + 8.6\,.$$

Proof. Define the function $\nu_r \colon x \mapsto \log_2 \log_2 x + \log_2 \log_2 \log_2 x + 6.6 - \log_2 x$ for $x \geq 4$. For $x \leq 4096$, we will prove the stronger statement that

$$\nu_r(x) \geq 0\,. \tag{8.14}$$

Note that for $x \leq 97 < 2^{6.6}$ and thus $\log_2 x < 6.6$, this is clearly fulfilled.

Thus, assume that $x \geq 98$. The derivative of $\nu_r(x)$ is

$$\begin{aligned}\frac{d}{dx}\nu_r(x) &= \frac{1}{\ln^2(2) x \log_2 x} + \frac{1}{\ln^3(2) x \log_2 x \log_2 \log_2 x} - \frac{1}{\ln(2) x} \\ &= \frac{\ln(2)\log_2\log_2 x + 1 - \ln^2(2)\log_2 x \log_2 \log_2 x}{\ln^3(2) x \log_2 x \log_2 \log_2 x}\,.\end{aligned}$$

This function is negative as its denominator is always positive and for its nominator, we have

$$\begin{aligned} & \ln(2)\log_2\log_2 x + 1 - \ln^2(2)\log_2 x\log_2\log_2 x \\ = \; & 1 + \ln(2)\log_2\log_2 x(1 - \ln(2)\log_2 x) \\ \overset{x \geq 98}{\leq} \; & 1 - 3.58\ln(2)\log_2\log_2 x \\ \overset{x \geq 98}{<} \; & 0\,.\end{aligned}$$

Thus, for $98 \leq x \leq 4096$, we have $\nu_r(x) \geq \nu_r(4096) \geq 0.02 > 0$. This proves Equation (*8.14*).

Now, we may assume that $4096 < x \leq 8192$, i.e., $12 < \log_2 x \leq 13$. This implies

$$\begin{aligned}\log_2 x + \log_2\log_2 x + \log_2\log_2\log_2 x + 8.6 \;&\overset{n>4096}{>}\; 12 + \log_2 12 + \log_2\log_2 12 + 8.6 \\ &>\; 26 \\ &\overset{x\leq 8192}{\geq}\; 2\lceil \log_2 x \rceil.\end{aligned}$$

This proves this lemma for all $4 \leq x \leq 8192$. □

For bounding the depth and size of our adder circuits, we need three more numerical lemmas. Each of them contains two statements as a preparation for Theorem 8.3.6 and Proposition 8.3.7, respectively.

Lemma 8.3.2. *For $x \geq 8192$ and $r \in \{1, 2\}$, the function*

$$\nu_r(x) := \frac{x}{\log_2^r x + 1}$$

is monotonely increasing in x.

Proof. We have

$$\frac{d}{dx}\left(\log_2^r x\right) = \frac{r(\log_2 x)^{r-1}}{\ln(2)x} \tag{8.15}$$

and thus

$$\frac{d}{dx}\nu_r(x) \overset{(8.15)}{=} \frac{\log_2^r x + 1 - x\frac{r(\log_2 x)^{r-1}}{\ln(2)x}}{\left(\log_2^r x + 1\right)^2} = \frac{\log_2^{r-1} x\left(\log_2 x - \frac{r}{\ln(2)}\right) + 1}{\left(\log_2^r x + 1\right)^2}.$$

As $r \leq 2$, the term $\log_2 x - \frac{r}{\ln 2}$ is surely positive for $x \geq 8192$, so $\frac{d}{dx}\nu_r(x) > 0$. □

Lemma 8.3.3. *For $x \geq 8192$ and $r \in \{2,3\}$, the function*

$$\nu_r(x) := \frac{(\log_2 x)^r}{x \log_2 \log_2 x}$$

is monotonely decreasing in x.

Proof. We have

$$\frac{d}{dx}\left(\log_2^r x\right) = \frac{r(\log_2 x)^{r-1}}{\ln(2)x}, \tag{8.16}$$

and

$$\frac{d}{dx}(x \log_2 \log_2 x) = \log_2 \log_2 x + x\frac{1}{\ln^2(2)x\log_2 x}. \tag{8.17}$$

Hence, the derivative of ν_r is

$$\begin{aligned}\frac{d}{dx}\nu_r(x) &\overset{(8.16),\,(8.17)}{=} \frac{\frac{r(\log_2 x)^{r-1}}{\ln(2)x}x\log_2\log_2 x - (\log_2 x)^r\left(\log_2\log_2 x + x\frac{1}{\ln^2(2)x\log_2 x}\right)}{x^2(\log_2\log_2 x)^2} \\ &= \frac{\frac{r(\log_2 x)^{r-1}\log_2\log_2 x}{\ln(2)} - (\log_2 x)^r\log_2\log_2 x - \frac{(\log_2 x)^{r-1}}{\ln^2(2)}}{x^2(\log_2\log_2 x)^2}.\end{aligned}$$

This is strictly negative for $x \geq 8192$ as

$$\begin{aligned}&\frac{r(\log_2 x)^{r-1}\log_2\log_2 x}{\ln(2)} - (\log_2 x)^r\log_2\log_2 x - \frac{(\log_2 x)^{r-1}}{\ln^2(2)} \\ <\ &(\log_2 x)^{r-1}\log_2\log_2 x\left(\frac{r}{\ln(2)} - \log_2 x\right) \\ \overset{\substack{r\leq 3,\\ x\geq 8192}}{<}\ &0.\end{aligned}$$

Hence, for $x \geq 8192$, we have $\frac{d}{dx}\nu_r(x) < 0$, which proves the statement. □

Lemma 8.3.4. *Let $k, l, n \in \mathbb{N}_{\geq 1}$ with $n \geq 8192$ and $l = \left\lceil \frac{n}{k} \right\rceil$ be given. We have*

$$\begin{aligned}&\log_2 l + \log_2\log_2 l + \log_2 k + \log_2\log_2(2k) + 8.16 \\ \leq\ &\log_2 n + \log_2\log_2 n + \log_2\log_2\log_2 n + \begin{cases} 8.6 & \text{if } k = \lceil \log_2 n \rceil, \\ 9.4 & \text{if } k = \left\lceil \log_2^2 n \right\rceil. \end{cases}\end{aligned}$$

Proof. **Case 1:** We have $k = \lceil \log_2 n \rceil$.
With $n \geq 8192$, we have

$$\frac{n}{k} \overset{k=\lceil \log_2 n \rceil}{\geq} \frac{n}{\log_2 n + 1} \overset{\substack{n \geq 8192, \\ \text{Lem. 8.3.2}}}{\geq} 585 \tag{8.18}$$

and thus

$$\log_2 l \overset{l=\lceil \frac{n}{k} \rceil}{\leq} \log_2\left(\frac{n}{k} + 1\right) \overset{(8.18)}{\leq} \log_2\left(\frac{n}{k}\right) + 0.003 = \log_2 n - \log_2 k + 0.003\,. \tag{8.19}$$

Furthermore, we have

$$\begin{aligned}
\log_2 \log_2 (2k) &\overset{k=\lceil \log_2 n \rceil}{\leq} \log_2\big(1 + \log_2(\log_2 n + 1)\big) \\
&\overset{n \geq 8192}{\leq} \log_2\left(1 + \log_2\left(\frac{14}{13} \log_2 n\right)\right) \\
&< \log_2(\log_2 \log_2 n + 1.11) \\
&\overset{n \geq 8192}{<} \log_2(1.3 \log_2 \log_2 n) \\
&< \log_2 \log_2 \log_2 n + 0.38\,.
\end{aligned} \tag{8.20}$$

Case 2: We have $k = \left\lceil \log_2^2 n \right\rceil$.
With $n \geq 8192$, we have

$$\frac{n}{k} \overset{k=\lceil \log_2 n \rceil}{\geq} \frac{n}{\log_2^2 n + 1} \overset{\substack{n \geq 8192, \\ \text{Lem. 8.3.2}}}{\geq} 48 \tag{8.21}$$

and thus

$$\log_2 l \overset{l=\lceil \frac{n}{k} \rceil}{\leq} \log_2\left(\frac{n}{k} + 1\right) \overset{(8.21)}{\leq} \log_2\left(\frac{n}{k}\right) + 0.03 = \log_2 n - \log_2 k + 0.03\,. \tag{8.22}$$

Furthermore, we have

$$\begin{aligned}
\log_2 \log_2 (2k) &\overset{k=\lceil \log_2 n \rceil}{\leq} \log_2\left(1 + \log_2\left(\log_2^2 n + 1\right)\right) \\
&\overset{n \geq 8192}{\leq} \log_2\left(1 + \log_2\left(\frac{170}{169} \log_2^2 n\right)\right) \\
&< \log_2(2 \log_2 \log_2 n + 1.009) \\
&\overset{n \geq 8192}{<} \log_2(2.28 \log_2 \log_2 n) \\
&< \log_2 \log_2 \log_2 n + 1.19\,.
\end{aligned} \tag{8.23}$$

For both cases together, from these inequalities, we conclude

$$\begin{aligned}
&\log_2 l + \log_2\log_2 l + \log_2 k + \log_2\log_2(2k) + 8.16\\
\overset{\substack{(8.19),\\(8.22)}}{\leq}\quad & \log_2 n - \log_2 k + \log_2\log_2 l + \log_2 k + \log_2\log_2(2k) + 8.16\\
+\quad & \begin{cases} 0.003 & \text{if } k = \lceil \log_2 n\rceil\\ 0.03 & \text{if } k = \left\lceil \log_2^2 n\right\rceil \end{cases}\\
\leq\quad & \log_2 n + \log_2\log_2 n + \log_2\log_2(2k) + \begin{cases} 8.163 & \text{if } k = \lceil \log_2 n\rceil\\ 8.19 & \text{if } k = \left\lceil \log_2^2 n\right\rceil \end{cases}\\
\overset{\substack{(8.20),\\(8.23)}}{<}\quad & \log_2 n + \log_2\log_2 n + \log_2\log_2\log_2 n + \begin{cases} 8.6 & \text{if } k = \lceil \log_2 n\rceil\\ 9.4 & \text{if } k = \left\lceil \log_2^2 n\right\rceil \end{cases}. \qquad \square
\end{aligned}$$

Lemma 8.3.5. *Let $k, l, n \in \mathbb{N}_{\geq 1}$ with $n \geq 8192$ and $l = \lceil \frac{n}{k} \rceil$ be given. If $k = \lceil \log_2 n\rceil$, then we have*

$$(6.6\log_2 l - 4)l \leq 6.6n\,.$$

If $k = \left\lceil \log_2^2 n\right\rceil$, then, we have

$$(6.6\log_2 l - 4)l \leq 0.51n\,.$$

Proof. **Case 1:** Assume that $k = \lceil \log_2 n\rceil$.

Here, we have

$$l = \left\lceil \frac{n}{k}\right\rceil \overset{k\geq \log_2 n}{\leq} \left\lceil \frac{n}{\log_2 n}\right\rceil \leq \frac{n}{\log_2 n} + 1 \tag{8.24}$$

and thus

$$\begin{aligned}
\log_2 l \overset{(8.24)}{\leq}\ & \log_2\left(\frac{n}{\log_2 n} + 1\right)\\
\overset{n\geq 8192}{\leq}\ & \log_2\left(1.002\frac{n}{\log_2 n}\right)\\
\leq\ & \log_2 n - \log_2\log_2 n + 0.003\,. \qquad (8.25)
\end{aligned}$$

Hence, we obtain

$$\begin{aligned}
(6.6\log_2 l - 4)l \overset{(8.25)}{\leq}\ & \big(6.6(\log_2 n - \log_2\log_2 n + 0.003) - 4\big)l\\
\overset{\substack{n\geq 8192,\\(8.24)}}{\leq}\ & \big(6.6(\log_2 n - \log_2\log_2 n + 0.003) - 4\big)\left(\frac{n}{\log_2 n} + 1\right)\\
\leq\ & 6.6n - \frac{6.6n}{\log_2 n}\log_2\log_2 n + \frac{0.02n}{\log_2 n}\\
& \quad + 6.6\log_2 n - 6.6\log_2\log_2 n + 0.02 - \frac{4n}{\log_2 n} - 4\\
<\ & 6.6n - \frac{6.6n}{\log_2 n}\log_2\log_2 n + 6.6\log_2 n\\
=\ & 6.6n + \frac{6.6n\log_2\log_2 n}{\log_2 n}\left(-1 + \frac{\log_2^2 n}{n\log_2\log_2 n}\right).
\end{aligned}$$

The last term can be bounded from above by $6.6n$ as

$$\frac{\log_2^2 n}{n \log_2 \log_2 n} \overset{\substack{n \geq 8192,\\ \text{Lem. } 8.3.3}}{\leq} \frac{\log_2^2(8192)}{8192 \log_2 \log_2(8192)} \leq 0.006 < 1\,.$$

This shows the first statement.

Case 2: Assume that $k = \left\lceil \log_2^2 n \right\rceil$.

We have

$$l = \left\lceil \frac{n}{k} \right\rceil \overset{k=\lceil \log_2^2 n \rceil}{\leq} \left\lceil \frac{n}{\log_2^2 n} \right\rceil \leq \frac{n}{\log_2^2 n} + 1\,. \tag{8.26}$$

and thus

$$\begin{aligned} \log_2 l &\overset{(8.26)}{\leq} \log_2\left(\frac{n}{\log_2^2 n} + 1\right) \\ &\overset{n \geq 8192}{\leq} \log_2\left(1.03 \frac{n}{\log_2^2 n}\right) \\ &\leq \log_2 n - 2 \log_2 \log_2 n + 0.05\,. \end{aligned} \tag{8.27}$$

This implies

$$\begin{aligned} (6.6 \log_2 l - 4) l &\overset{(8.27)}{\leq} \big(6.6(\log_2 n - 2 \log_2 \log_2 n + 0.05) - 4\big) l \\ &\overset{\substack{n \geq 8192,\\ (8.26)}}{\leq} \big(6.6(\log_2 n - 2 \log_2 \log_2 n + 0.05) - 4\big) \left(\frac{n}{\log_2^2 n} + 1\right) \\ &\leq 6.6 \frac{n}{\log_2 n} - \frac{13.2n}{\log_2^2 n} \log_2 \log_2 n + \frac{0.33n}{\log_2^2 n} \\ &\quad + 6.6 \log_2 n - 13.2 \log_2 \log_2 n + 0.33 - \frac{4n}{\log_2^2 n} - 4 \\ &< 6.6 \frac{n}{\log_2 n} - \frac{13.2n}{\log_2^2 n} \log_2 \log_2 n + 6.6 \log_2 n \\ &\overset{n \geq 8192}{\leq} 0.51n + \frac{n \log_2 \log_2 n}{\log_2^2 n} \left(-13.2 + 6.6 \frac{\log_2^3 n}{n \log_2 \log_2 n}\right). \end{aligned}$$

The last term can be bounded from above by $0.51n$ as

$$6.6 \frac{\log_2^3 n}{n \log_2 \log_2 n} \overset{\substack{n \geq 8192,\\ \text{Lem. } 8.3.3}}{\leq} 6.6 \frac{\log_2^3(8192)}{8192 \log_2 \log_2(8192)} \leq 0.08 < 13.2\,.$$

This proves this lemma. □

Using the previous lemmas, we now prove the main theorem of this chapter.

Theorem 8.3.6. *Let $n \in \mathbb{N}$ with $n \geq 4$ be given. In running time $\mathcal{O}(n \log_2 n)$, we can construct an adder circuit C_n^2 on n input pairs with depth*

$$d(C_n^2) \leq \log_2 n + \log_2 \log_2 n + \log_2 \log_2 \log_2 n + 8.6$$

and size

$$s(C_n^2) \leq 22.7n\,.$$

Proof. If $4 \leq n \leq 8192$, we use the adder circuit by Ladner and Fischer [LF80] with size at most $12n$ and depth at most $2\lceil \log_2 n \rceil$, see Theorem 2.6.24 with $f = 0$. Its size is sufficiently small because $22.7 > 12$; and its depth is sufficiently small by Lemma 8.3.1.

Hence, assume that $n \geq 8192$. We apply the linearization from Theorem 8.2.4 to the adder family $(B_l)_{l \in \mathbb{N}} := \left(A^1_l\right)_{l \in \mathbb{N}}$ from Theorem 8.1.3 with $k = \lceil \log_2 n \rceil$ and thus $l = \left\lceil \frac{n}{k} \right\rceil$. For the adder family $(A_k)_{k \in \mathbb{N}}$, we use the circuits from Proposition 8.2.5. For the And-prefix circuit family $(S_k)_{k \in \mathbb{N}}$, we use the Ladner-Fischer [LF80] circuits $\left(S^3_k\right)_{k \in \mathbb{N}}$ from Theorem 2.6.24 with $f = 3$, which is viable as $\lceil \log_2 k \rceil \geq \log_2 \log_2 n \overset{n \geq 8192}{\geq} 3.7$. This leads to a size of $s(S_k) \leq 2\left(1 + \frac{1}{2^3}\right)k = 2.25k$. For the And-Or paths AOP_k, we use the circuit from Corollary 3.4.21 (with at most $2k$ inputs). Thus, we have

$$\begin{aligned}
d(A_k) &\leq k + 2\,, & s(A_k) &\leq 3.5k\,,\\
d(B_l) &\leq \log_2 l + \log_2 \log_2 l + 3.58\,, & s(B_l) &\leq 6.6 l \log_2 l\,,\\
d(AOP_k) &\leq \log_2(2k) + \log_2 \log_2(2k) + 1.58\,, & s(AOP_k) &\leq 8.3k - 4\,,\\
d(S_k) &\leq \lceil \log_2 k \rceil + 3\,, & s(S_k) &\leq 2.25k\,.
\end{aligned} \tag{8.28}$$

We denote the adder circuit resulting from Theorem 8.2.4 applied with this setting by A^2_n.

We now bound the depth of A^2_n. Since $k \overset{n \geq 8192}{\geq} 13$, we have

$$\begin{aligned}
\log_2(2k) + \log_2 \log_2(2k) + 1.58 &\overset{k \geq 13}{\geq} \log_2 k + \log_2 \log_2(26) + 2.58\\
&\geq \log_2 k + 2.23 + 2.58\\
&\geq \lceil \log_2 k \rceil + 3\\
&= d(S_k)\,.
\end{aligned} \tag{8.29}$$

Furthermore, we have

$$\begin{aligned}
& \log_2 l + \log_2 \log_2 l + \log_2(2k) + \log_2 \log_2(2k) + 5.16\\
> \; & \log_2 l + \log_2 k + 6.16\\
\overset{l = \left\lceil \frac{n}{k} \right\rceil}{\geq} \; & \log_2 \log_2 n + 6.16\\
\overset{k = \lceil \log_2 n \rceil}{\geq} \; & k + 5.16\\
\overset{(8.28)}{=} \; & d(A_k) + 3.16\,.
\end{aligned} \tag{8.30}$$

Therefore, the depth of our adder circuit can be bounded by

$$\begin{aligned}
d(A_n^2) &\overset{\text{Thm. 8.2.4}}{\leq} \max\left\{ d(B_l) + \max\left\{d(AOP_k), d(S_k)\right\} + 2,\ d(A_k) + 1 \right\} \\
&\overset{\substack{(8.28),\\(8.29),\\(8.30)}}{=} \log_2 l + \log_2\log_2 l + 3.58 + \log_2(2k) + \log_2\log_2(2k) + 1.58 + 2 \\
&= \log_2 l + \log_2\log_2 l + \log_2 k + \log_2\log_2(2k) + 8.16 \\
&\overset{\text{Lem. 8.3.4}}{\leq} \log_2 n + \log_2\log_2 n + \log_2\log_2\log_2 n + 8.6\,.
\end{aligned}$$

The size of A_n^2 can be bounded by

$$\begin{aligned}
s(A_n^2) &\overset{\text{Thm. 8.2.4}}{\leq} \sum_{j=0}^{l-1} s(A_{n_j}) + \sum_{j=1}^{l-1} s(S_{n_j}) + \sum_{j=1}^{l-1} s(AOP_{n_j}) + s(B_l) + 2n \\
&\overset{(8.28)}{<} \sum_{j=0}^{l-1} (3.5n_j + 2.25n_j + 8.3n_j - 4) + 6.6l\log_2 l + 2n \\
&= 16.05n + (6.6\log_2 l - 4)l \\
&\overset{\text{Lem. 8.3.5}}{\leq} 16.05n + 6.6n \\
&< 22.7n\,.
\end{aligned}$$

For proving the running time, it suffices to consider the case $n > 8192$. Then, we construct adder circuits B_l on $l = \lceil \frac{n}{k} \rceil$ inputs using Theorem 8.1.3, which can be executed in time $\mathcal{O}(l\log_2^2 l) = \mathcal{O}(n\log_2 n)$ (see Theorem 8.1.3). For the construction of the l And-Or paths AOP_k, we apply the algorithm from Corollary 3.4.21 with a running time of $\mathcal{O}(k\log_2 k)$ for each And-Or path, which yields a running time of $\mathcal{O}(lk\log_2 k) = \mathcal{O}(n\log_2\log_2 n)$ for constructing all these And-Or paths. The remaining part of the algorithm (including the computation of the And-prefix circuit family S (see Proposition 2.6.25) and adder family A (see Proposition 8.2.5) can be done in linear time.

Thus, we get an overall running time of $\mathcal{O}(n\log_2 n)$. □

These circuits are the fastest linear-size adder circuits known so far. With a depth increase of only 0.8, we can reduce the size significantly as follows.

Proposition 8.3.7. *Let $n \in \mathbb{N}$ with $n \geq 4$ be given. In running time $\mathcal{O}(n\log_2\log_2 n)$, we can construct an adder circuit C_n^3 on n input pairs with depth*

$$d(C_n^3) \leq \log_2 n + \log_2\log_2 n + \log_2\log_2\log_2 n + 9.4$$

and size

$$s(C_n^3) \leq 17.6n\,.$$

Proof. Consider again the proof of Theorem 8.3.6. We proceed similarly, but choose k differently and use different adder circuits A_k.

If $4 \leq n \leq 8192$, we again use the adder circuit by Ladner and Fischer [LF80] with size at most $12n$ and depth at most $2\lceil \log_2 n\rceil$, see Theorem 2.6.24 with $f = 0$.

Its size is sufficiently small because $17.6 > 12$; and its depth is sufficiently small by Lemma 8.3.1.

Thus, assume that $n \geq 8192$. We again apply the linearization from Theorem 8.2.4 with B_l and AOP_k chosen as in Theorem 8.3.6, but with S_k and A_k both computed by the Ladner-Fischer parallel prefix circuit from [LF80] with $f = 3$, see also Theorem 2.6.24. This way, we obtain

$$\begin{aligned} d(A_k) &\leq 2\big(\lceil \log_2 k \rceil + 3\big) \\ d(S_k) &\leq \lceil \log_2 k \rceil + 3 \\ s(A_k) + s(S_k) &\leq 6.75k\,, \end{aligned} \tag{8.31}$$

and the bounds regarding B_l and AOP_k from Equation (8.28) still hold.

This time, we choose $k = \left\lceil \log_2^2 n \right\rceil$ and $l = \left\lceil \frac{n}{k} \right\rceil$. Note that the choice of f is valid as

$$\log_2 k \geq \log_2(\log_2^2 n) = 2 \log_2 \log_2 n \overset{n \geq 8192}{>} 7 > 3\,.$$

Denote the arising adder circuit by A_n^3.

Note that Equation (8.29) is still valid as k is larger than before. For bounding the depth of A_k, we calculate

$$\begin{aligned} 2\left\lceil \log_2 \left\lceil \log_2^2 n \right\rceil \right\rceil &\leq 2\left\lceil \log_2 \left(1 + \log_2^2 n\right) \right\rceil \\ &\overset{n \geq 8192}{\leq} 2\left\lceil \log_2 \left(1.006 \log_2^2 n\right) \right\rceil \\ &< 2\lceil 0.009 + 2 \log_2 \log_2 n \rceil \\ &< 4 \log_2 \log_2 n + 2.018 \end{aligned} \tag{8.32}$$

and thus

$$\begin{aligned} d(A_k) &\overset{\substack{(8.31),\\ k=\lceil \log_2^2 n \rceil}}{\leq} 2\left\lceil \log_2 \left\lceil \log_2^2 n \right\rceil \right\rceil + 6 \\ &\overset{(8.32)}{<} 4 \log_2 \log_2 n + 8.018 \\ &\overset{n \geq 5315}{\leq} \log_2 n + 10.16 \\ &\overset{l=\lceil \frac{n}{k} \rceil}{\leq} \log_2 l + \log_2 k + 10.16 \\ &\overset{k \geq 169}{<} \log_2 l + \log_2 \log_2 l + \log_2(2k) + \log_2 \log_2(2k) + 6.16\,. \end{aligned} \tag{8.33}$$

Hence, the depth of A_n^3 can be bounded by

$$\begin{aligned} d(A_n^3) &\overset{\text{Thm. 8.2.4}}{\leq} \max\left\{ d(B_l) + \max\left\{ d(AOP_k), d(S_k) \right\} + 2,\ d(A_k) + 1 \right\} \\ &\overset{\substack{(8.28),\\ (8.29),\\ (8.32)}}{=} \log_2 l + \log_2 \log_2 l + 3.58 + \log_2(2k) + \log_2 \log_2(2k) + 1.58 + 2 \\ &= \log_2 l + \log_2 \log_2 l + \log_2 k + \log_2 \log_2(2k) + 8.16 \\ &\overset{\text{Lem. 8.3.4}}{\leq} \log_2 n + \log_2 \log_2 n + \log_2 \log_2 \log_2 n + 9.4\,. \end{aligned}$$

For the size of A_n^3, we obtain

$$\begin{aligned}
s(A_n^2) &\overset{\text{Thm. 8.2.4}}{\leq} \sum_{j=0}^{l-1} s(A_{n_j}) + \sum_{j=1}^{l-1} s(S_{n_j}) + \sum_{j=1}^{l-1} s(AOP_{n_j}) + s(B_l) + 2n \\
&\overset{\substack{(8.31),\\(8.28)}}{<} \sum_{j=0}^{l-1}(6.75n_j + 8.3n_j - 4) + 6.6l\log_2 l + 2n \\
&= 17.05n + (6.6\log_2 l - 4)l \\
&\overset{\text{Lem. 8.3.5}}{\leq} 17.05n + 0.51n \\
&< 17.6n\,.
\end{aligned}$$

For proving the running time, it again suffices to consider the case $n > 8192$. For $k = \left\lceil \log_2^2 n \right\rceil$, constructing the adder circuits B_l using Theorem 8.1.3 takes time $\mathcal{O}(l\log_2^2 l) = \mathcal{O}(n)$ (see Theorem 8.1.3). For the construction of the l AND-OR paths AOP_k using Corollary 3.4.21, we need a running time of $\mathcal{O}(lk\log_2 k) = \mathcal{O}(n\log_2\log_2 n)$. Again, the remaining part of the algorithm (including the computation of A_k and S_k (see Proposition 2.6.25) can be done in linear time.

Thus, we get an overall running time of $\mathcal{O}(n\log_2\log_2 n)$. □

Summary

We have considered the problems of designing algorithms for the computation of fast circuits both for And-Or paths and for binary addition.

A measure for the speed of a circuit which is often considered in the literature is circuit depth. However, on a computer chip, it is more realistic to assume that the input signals are available at individual prescribed arrival times. Circuit delay is natural a generalization of circuit depth to arrival times. In the following, let $n \in \mathbb{N}$ be the number of inputs, and in case arrival times are present, denote them by $a_0, \dots, a_{n-1} \in \mathbb{N}$.

We have improved the best known upper bounds on the optimum depth and delay significantly both for adder and And-Or path circuits.

Regarding depth optimization of And-Or path circuits, we have proposed the first polynomial-time algorithm constructing circuits which have a linear size and, at the same time, a depth of $\log_2 n + \log_2 \log_2 n + \text{const}$. By Commentz-Walter [Com79], this is the best possible asymptotic depth guarantee for And-Or paths and for adder circuits. For adder circuits, we achieve such a depth bound with a size of $\mathcal{O}(n \log_2 n)$. Furthermore, we have introduced an algorithm constructing linear-size adder circuits with a depth of $\log_2 n + \log_2 \log_2 n + \log_2 \log_2 \log_2 n + \text{const}$, improving the gap to the lower bound from $\mathcal{O}\left(\sqrt{\log_2 n}\right)$ to $\mathcal{O}(\log_2 \log_2 \log_2 n)$. Both the And-Or path and the adder optimization algorithm have a running time of $\mathcal{O}(n \log_2 n)$.

For delay optimization of And-Or path circuits, we have presented an algorithm with running time $\mathcal{O}(n \log_2^2 n)$ with a delay guarantee of $\log_2 W + \log_2 \log_2 n + \log_2 \log_2 \log_2 n + \text{const}$, where $W = \sum_{i=0}^{n-1} 2^{a_i}$. This significantly improves upon the previously best delay of $\lceil \log_2 W \rceil + 2\sqrt{2 \log_2 n} + \text{const}$ in comparison to the lower bound of $\lceil \log_2 W \rceil$.

Moreover, we have presented the first exact delay optimization algorithm for generalized And-Or paths. Before, there were only exact algorithms for depth optimization of And-Or paths. Our algorithm has a theoretical running time of $\mathcal{O}(3^n)$, and for And-Or paths even only $\mathcal{O}\left(\left(\sqrt{6}\right)^n\right)$. For the special case of depth optimization of And-Or paths, the running time is improved to $\mathcal{O}(n2.02^n)$. For this case, an exact algorithm with running time $\mathcal{O}\left(\left(\sqrt{6}\right)^n\right)$ was known before.

Using efficient pruning techniques, we have shown that we can drastically improve our empirical running time. This way, we have computed depth-optimum And-Or path circuits for up to 64 inputs, while the previously best exact algorithm could only solve instances with up to 29 inputs due to running time issues. Running our algorithm takes up to 1.5 seconds on any of these instances with up to 60 inputs, and less than 3 hours on the remaining 4 instances. For And-Or path instances with non-uniform input arrival times, our average running time on a set of instances with 60 inputs is less than 2 minutes, but the maximum running time is 24 minutes.

Together with one of our theoretical results, our computations yield the optimum depths of n-bit adder circuits for all $n = 2^k$ with $k \leq 13$, which – to the best of our knowledge – have not been known before for any $n \geq 32$.

For practical applications, we have proposed a dynamic program with running time $\mathcal{O}(n^4)$ which fulfills the best known theoretical delay guarantee, but is also very efficient and computes very good solutions in practice. On an artificial testbed, we have demonstrated that this algorithm outperforms the previously best implemented algorithms. Moreover, using our exact algorithm, we computed the optimum delay achievable on each instance of this testbed. We have observed that our practical algorithm computes a delay-optimum solution on 95% of these instances, while the previously best known algorithms compute optimum circuits only for 10% of the instances.

We have presented a logic optimization tool for chip design called BonnLogic which optimizes the logical structure of the most timing-critical paths with our practical algorithm as a core routine. BonnLogic is successfully applied in the chip design flow of IBM. We have demonstrated the effectiveness and efficiency of BonnLogic on recent real-world chips.

Index

NOTATION

C^*	Dual of a circuit C. Definition 2.2.10.
C_ϕ	Formula circuit corresponding to a Boolean formula ϕ. Definition 2.2.7.
C_v	Circuit subordinate to a vertex v of circuit C. Definition 2.2.9.
$C\,\|_{x_i=\alpha}$	Reduced circuit of a circuit C with respect to input x_i and $\alpha \in \{0,1\}$. Definition 2.3.13.
$d(C)$	Depth of a circuit C. Chapter 8.
$d_{\min}(n,m)$	For $n, m \in \mathbb{N}$, we have $d_{\min}(n,m) = \min\big\{\, d \in \mathbb{N} : m \leq \mu(d,n) \,\big\}$. Definition 3.1.7.
$\operatorname{delay}(C)$	Delay of a circuit C with input arrival times from the context. Definition 2.3.2.
$\operatorname{delay}(C;a)$	Delay of a circuit C with input arrival times a. Definition 2.3.2.
$\operatorname{delay}(\phi)$	Delay of a Boolean formula ϕ with input arrival times from the context. Definition 2.3.8.
$\operatorname{delay}(\phi;a)$	Delay of a Boolean formula ϕ with input arrival times a. Definition 2.3.8.
$\operatorname{depth}(C)$	Depth of a circuit C. Definition 2.3.1.
$\operatorname{depth}(\phi)$	Depth of a Boolean formula ϕ. Definition 2.3.8.
$\mathcal{E}(C)$	Edges of a circuit C. Definition 2.2.2.
f^*	Dual of the Boolean function f. Definition 2.1.29.
f_C	Boolean function corresponding to a circuit C. Definition 2.2.2.
f_ϕ	Boolean function that realizes a Boolean formula ϕ. Definition 2.1.8.
$f(s,t)$	Extended AND-OR path $f(s,t) = s_0 \wedge \ldots \wedge s_{n-1} \wedge g(t)$ with symmetric inputs $s = (s_0, \ldots, s_{n-1})$ and alternating inputs $t = (t_0, \ldots, t_{m-1})$. Definition 2.6.14.
$f\,\|_{x_i=\alpha}$	Restriction of a Boolean function f to $x_i = \alpha \in \{0,1\}$. Definition 2.1.3.
$\operatorname{fanout}(C)$	Maximum fanout of any vertex of a circuit C. Definition 2.3.3.
$\operatorname{fanout}(v)$	Fanout $\operatorname{fanout}(v) = \|\delta^+(v)\|$ of a vertex v in a circuit C. Definition 2.2.9.
$f_{i,j,k}$	Extended AND-OR path $f_{i,j,k} = f\big((t_i, t_{i+2}, \ldots, t_{j-4}, t_{j-2}), (t_j, \ldots, t_k)\big)$. Notation 6.1.1.
$\operatorname{flodd}(x)$	Given $x \in \mathbb{R}$, we write $\operatorname{flodd}(x) = \max\{\, y \in \mathbb{Z} : y \text{ odd}, y \leq x \,\}$. Notation 3.4.15.

$\mathcal{G}(C)$	Gates of a circuit C. Definition 2.2.2.
$g(t)$	AND-OR path $g(t) = t_0 \wedge (t_1 \vee (t_2 \wedge \dots))$ with Boolean input variables $t = (t_0, \dots, t_{m-1})$. Definition 2.5.1.
$\mathrm{gt}(C)$	Gate type of $\mathrm{out}(C)$ for an undetermined circuit C. Definition 6.1.7.
$\mathrm{gt}(v)$	Gate type of a gate v in a circuit C. Definition 2.2.2.
$h(t;\Gamma)$	A generalized AND-OR path $h(t;\Gamma) = t_0 \circ_0 \big(t_1 \circ_1 (\dots)\big)$ on inputs $t = (t_0, \dots, t_{m-1})$ with gates $\Gamma = (\circ_0, \dots, \circ_{m-2})$. Definition 2.5.5.
$h(t;\Gamma)_{[i:j]}$	Restriction of a generalized AND-OR path $h(t;\Gamma)$ to the input range from t_i up to t_j for $i \leq j$. Notation 5.2.1.
$h(t;\Gamma)_{\widehat{S}}$	Generalized AND-OR path arising from $h(t;\Gamma)$ by deleting the input set S. Notation 5.2.1.
$h(t;\Gamma)_{S_k^\circ}$	Generalized AND-OR path $h(t;\Gamma)_{S_k^\circ} = \Big(h(t;\Gamma)_{[0:i_k]}\Big)_{\widehat{S_2^\circ}}$. Notation 5.2.1.
$h(t;\Gamma)_{\widehat{t_i}}$	Generalized AND-OR path arising from $h(t;\Gamma)$ by deleting the input t_i. Notation 5.2.1.
$\mathcal{I}(C)$	Inputs of a circuit C. Definition 2.2.2.
$\mathcal{I}_v(C)$	Inputs in the input cone of a vertex v of a circuit C. Definition 2.2.9.
$\mathrm{lit}(\iota)$	Set of literals of an implicant ι of a Boolean function f. Definition 2.1.19.
$m(d,n)$	Capacity of d and n for $d, n \in \mathbb{N}$. Maximum number m of alternating inputs such that an extended AND-OR path with n symmetric inputs and m alternating inputs can be realized with depth d. Definition 3.1.1.
$\mathbb{N}$	Set of natural numbers including 0. Section 2.1.
$\mathcal{O}(C)$	Outputs of a circuit C. Definition 2.2.2.
$\mathrm{out}(C)$	Unique output gate of a circuit C. Definition 2.2.2.
$\mathrm{out}_i(A)$	Output gate of an adder circuit A which computes the ith carry bit. Definition 2.4.4.
$P_0 \mathbin{+\!\!\!+} \dots \mathbin{+\!\!\!+} P_c$	signal partition of a generalized AND-OR path with input parts $P_0, \dots, P_c$. Definition 2.5.6.
$\mathrm{PI}(f)$	Set of all prime implicants of a Boolean function f. Definition 2.1.19.
S°	Set of all same-gate inputs of a generalized AND-OR path $h(t;\Gamma)$ and a gate type $\circ \in \{\text{AND}, \text{OR}\}$. Notation 5.2.1.
$s(C)$	Size, i.e., number of gates, of a circuit C. Chapter 8.
$\mathrm{size}(C)$	Number of gates of a circuit C. Definition 2.3.4.
$\mathrm{size}(\phi)$	Number of gates of a Boolean formula ϕ. Definition 2.3.8.
$\mathrm{sym}(x)$	Logical multiple-input AND function $x = (x_0, \dots, x_{n-1}) \mapsto x_0 \wedge \dots \wedge x_{n-1}$. Definition 2.3.20.
$\mathcal{V}(C)$	Vertices of a circuit C. Definition 2.2.2.
$\mathcal{V}_v(C)$	Input cone of a vertex v of a circuit C. Definition 2.2.9.
$W(C)$	Weight $\sum_{v \in \delta^{-1}(\mathrm{out}(C))} 2^{a(v)}$ of an undetermined circuit C with arrival times a from the context. Definition 6.1.8.
$W(t)$	Weight $W(t) = W(t;a)$ with arrival times $a(t_0), \dots, a(t_{m-1}) \in \mathbb{N}$ from the context. Definition 2.3.16.
$W(t;a)$	Weight $W(t) = \sum_{i=0}^{n-1} 2^{a(t_i)}$ of inputs $t = (t_0, \dots, t_{m-1})$ with arrival times $a(t_0), \dots, a(t_{m-1}) \in \mathbb{N}$. Definition 2.3.16.

$\mu(d,n)$	Given $d,n \in \mathbb{N}$, we have $\mu(d,n) = \frac{2^d-n-2}{d} + 2$. Definition 3.1.5.
$\nu(d,w)$	Given $d,w \in \mathbb{N}$, we have $\nu(d,w) = \zeta \frac{2^{d-1}-w}{d \log_2 d}$. Definition 4.1.1.
ϕ^*	Dual of the Boolean formula ϕ. Definition 2.1.28.
ϕ_C	Boolean formula corresponding to a circuit C. Definition 2.2.2.
Ω_{mon}	Standard monotone basis $\Omega_{\text{mon}} = \{\textsc{And2}, \textsc{Or2}\}$. Definition 2.2.3.
Ω_{nmon}	Standard non-monotone basis $\Omega_{\text{nmon}} = \{\textsc{And2}, \textsc{Or2}, \textsc{Not}\}$. Definition 2.2.3.
$\cdot \wedge \cdot$	Logical AND operation. Definition 2.1.6.
$\bigwedge \cdot$	Logical multiple-input AND function. Remark 2.1.17.
$\cdot \vee \cdot$	Logical OR operation. Definition 2.1.6.
$\bigvee \cdot$	Logical multiple-input OR function. Remark 2.1.17.
$\cdot \oplus \cdot$	Logical XOR operation. Definition 2.1.5.
$\overline{\cdot}$	Logical NOT operation. Definition 2.1.6.
$\cdot \circ \cdot$	A prefix operator. Section 2.6.3.
$\cdot \circ_{\text{p}} \cdot$	Adder prefix operator. Definition 2.6.22.
$\widehat{t}$	Input variables arising from input variables t by deleting every other variable as in Definition 2.6.15.
$\cdot \mathbin{+\!\!\!+} \cdot$	Concatenation of two tuples. For example, $(w,x) \mathbin{+\!\!\!+} (y,z) = (w,x,y,z)$. Section 2.6.2.

List of Tables

List of Algorithms

Bibliography

[AGD15] Luca Amarú, Pierre-Emmanuel Gaillardon, and Giovanni De Micheli. "The EPFL combinational benchmark suite". In: *Proceedings of the 24th International Workshop on Logic & Synthesis.* 2015 (cit. on p. 204).

[Alp+06] Charles J. Alpert, Jiang Hu, Sachin S. Sapatnekar, and C. N. Sze. "Accurate estimation of global buffer delay within a floorplan". In: *Transactions on Computer-Aided Design of Integrated Circuits and Systems* 25.6 (2006), pp. 1140–1145 (cit. on pp. 193, 200).

[Ama+17] Luca Amarú, Mathias Soeken, Patrick Vuillod, Jiong Luo, Alan Mishchenko, Pierre-Emmanuel Gaillardon, Janet Olson, Robert Brayton, and Giovanni De Micheli. "Enabling exact delay synthesis". In: *Proceedings of the 2017 International Conference on Computer-Aided Design.* 2017, pp. 352–359 (cit. on p. 196).

[AMS08] Charles J. Alpert, Dinesh P. Mehta, and Sachin S. Sapatnekar, eds. *Handbook of Algorithms for Physical Design Automation.* 2008 (cit. on p. 195).

[Bar+06] Christoph Bartoschek, Stephan Held, Dieter Rautenbach, and Jens Vygen. "Efficient generation of short and fast repeater tree topologies". In: *Proceedings of the 2006 International Symposium on Physical Design.* 2006, pp. 120–127 (cit. on p. 200).

[Bar+09] Christoph Bartoschek, Stephan Held, Dieter Rautenbach, and Jens Vygen. "Fast buffering for optimizing worst slack and resource consumption in repeater trees". In: *Proceedings of the 2007 International Symposium on Physical Design.* 2009, pp. 43–50 (cit. on p. 202).

[Bar+10] Christoph Bartoschek, Stephan Held, Jens Maßberg, Dieter Rautenbach, and Jens Vygen. "The repeater tree construction problem". In: *Information Processing Letters* 110.24 (2010), pp. 1079–1083 (cit. on p. 32).

[BH19] Ulrich Brenner and Anna Hermann. "Faster carry bit computation for adder circuits with prescribed arrival times". In: *ACM Transactions on Algorithms* 15.4 (2019), Art. No. 45, 45:1–45:23 (cit. on pp. 10, 11, 103).

[BH20] Ulrich Brenner and Anna Hermann. *Delay optimization of combinational logic by AND-OR path restructuring.* Technical Report 201198. Research Institute for Discrete Mathematics, University of Bonn, 2020 (cit. on pp. 10, 11, 159, 166, 191, 197).

[BK82] R. P. Brent and H. T. Kung. “A regular layout for parallel adders”. In: *Transactions on Computers* 3 (1982), pp. 260–264 (cit. on p. 54).

[BM10] Robert Brayton and Alan Mishchenko. “ABC: An academic industrial-strength verification tool”. In: *International Conference on Computer Aided Verification*. Springer. 2010, pp. 24–40 (cit. on p. 196).

[Bre+15] Ulrich Brenner, Anna Hermann, Nils Hoppmann, and Philipp Ochsendorf. “BonnPlace: A Self-Stabilizing Placement Framework”. In: *Proceedings of the 2015 International Symposium on Physical Design*. 2015, pp. 9–16 (cit. on p. 202).

[CG15] Willem Conradie and Valentin Goranko. *Logic and Discrete Mathematics: A Concise Introduction*. 2015 (cit. on p. 147).

[CH11] Yves Crama and Peter L. Hammer. *Boolean Functions: Theory, Algorithms, and Applications*. 2011 (cit. on pp. 13, 19, 20, 195).

[Cha+06] S. Chatterjee, A. Mishchenko, R. Brayton, X. Wang, and T. Kam. “Reducing structural bias in technology mapping”. In: *Transactions on Computer-Aided Design of Integrated Circuits and Systems* 25.12 (2006), pp. 2894–2903 (cit. on p. 196).

[Cho04] Youngmoon Choi. “Parallel Prefix Adder Design”. PhD thesis. The University of Texas at Austin, 2004 (cit. on p. 55).

[Com79] Beate Commentz-Walter. “Size-depth tradeoff in monotone Boolean formulae”. In: *Acta Informatica* 12.3 (1979), pp. 227–243 (cit. on pp. 9, 10, 13, 42–44, 57, 61, 75, 103, 138, 205, 206, 227).

[Cor03] Jordi Cortadella. “Timing-driven logic bi-decomposition”. In: *Transactions on Computer-Aided Design of Integrated Circuits and Systems* 22.6 (2003), pp. 675–685 (cit. on p. 196).

[Cou94] Olivier Coudert. “Two-level logic minimization: an overview”. In: *Integration. The VLSI Journal* 17.2 (1994), pp. 97–140 (cit. on p. 195).

[CP92] Kamal Chaudhary and Massoud Pedram. “A near optimal algorithm for technology mapping minimizing area under delay constraints”. In: *Proceedings of the 29th Design Automation Conference*. Vol. 8. 12. 1992, pp. 492–498 (cit. on p. 196).

[CS80] Beate Commentz-Walter and Juergen Sattler. “Size-depth tradeoff in non-monotone Boolean formulae”. In: *Acta Informatica* 14.3 (1980), pp. 257–269 (cit. on p. 42).

[DGK94] Srinivas Devadas, Abhijit Ghosh, and Kurt Keutzer. *Logic Synthesis*. 1994 (cit. on p. 196).

[Elb17] Lucas Elbert. “Aproximationsalgorithmen im Technology Mapping”. German. BA thesis. University of Bonn, 2017 (cit. on pp. 196, 199, 201).

[Elm48] W. C. Elmore. “The transient response of damped linear networks with particular regard to wideband amplifiers”. In: *Journal of applied physics* 19.1 (1948), pp. 55–63 (cit. on p. 193).

[Ges+13] Michael Gester, Dirk Müller, Tim Nieberg, Christian Panten, Christian Schulte, and Jens Vygen. “BonnRoute: Algorithms and Data Structures for Fast and Good VLSI Routing”. In: *Transactions on Design Automation of Electronic Systems* 18.2 (2013), Art. No. 32 (cit. on p. 195).

[GGS07] S. B. Gashkov, M. I. Grinchuk, and I. S. Sergeev. “Circuit design of an adder of small depth”. In: *Diskretnyi Analiz I Issledovanie Operatsii* 14.1 (2007), pp. 27–44. English translation: S. B. Gashkov, M. I. Grinchuk, and I. S. Sergeev. “Circuit design of an adder of small depth”. In: *Journal of Applied and Industrial Mathematics* 2.2 (2008), pp. 167–178 (cit. on pp. 56, 217).

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractibility: A Guide to the Theory of NP-Completeness*. 1979 (cit. on p. 195).

[Gol76] Martin C. Golumbic. “Combinatorial merging”. In: *Transactions on Computers* 25.11 (1976), pp. 1164–1167 (cit. on pp. 28, 29, 31, 32).

[Gri08] M. I. Grinchuk. “Sharpening an upper bound on the adder and comparator depths”. In: *Diskretnyi Analiz I Issledovanie Operatsii* 15.2 (2008), pp. 12–22. English translation: M. I. Grinchuk. “Sharpening an upper bound on the adder and comparator depths”. In: *Journal of Applied and Industrial Mathematics* 3.1 (2009), pp. 61–67 (cit. on pp. 9, 46, 56, 58, 61–63, 75, 103–105).

[Gri13] Mikhail I. Grinchuk. “Low depth circuit design”. US patent 8499264 B2. 2013 (cit. on pp. 57, 144, 157, 158).

[Heg18] Falko Hegerfeld. “Optimal Monotone Realizations of And-Or-Paths”. MA thesis. University of Bonn, 2018 (cit. on pp. 10, 57, 131, 144, 152–154, 157, 158).

[Hel+11] Stephan Held, Bernhard Korte, Dieter Rautenbach, and Jens Vygen. “Combinatorial optimization in VLSI design.” In: *Combinatorial Optimization-Methods and Applications* 31 (2011), pp. 33–96 (cit. on p. 191).

[Hel08] Stephan Held. “Timing Closure in Chip Design”. PhD thesis. University of Bonn, 2008 (cit. on p. 195).

[Hel09] Stephan Held. “Gate sizing for large cell-based designs”. In: *Proceedings of the 2009 Conference on Design, Automation and Test in Europe* (Nice, France). 2009, pp. 827–832 (cit. on p. 202).

[Hit18] Jan Maik Hitzschke. “Untere Schranken für Tiefe und Delay von AND-OR-Pfaden”. German. Bachelor’s thesis. University of Bonn, 2018 (cit. on pp. 44, 57, 75).

[HS06] Gary D. Hachtel and Fabio Somenzi. *Logic Synthesis and Verification Algorithms*. 2006 (cit. on p. 196).

[HS17a] Stephan Held and Sophie Theresa Spirkl. “Binary adder circuits of asymptotically minimum depth, linear size, and fan-out two”. In: *ACM Transactions on Algorithms* 14.1 (2017), pp. 1–18 (cit. on pp. 10, 56, 206, 217).

[HS17b] Stephan Held and Sophie Theresa Spirkl. “Fast prefix adders for non-uniform input arrival times”. In: *Algorithmica* 77.1 (2017), pp. 287–308 (cit. on pp. 10, 55, 56, 58–60, 118, 124, 125, 128, 159, 164, 165, 177–185, 187, 209).

[HSC82] Robert B. Hitchcock, Sr., Gordon L. Smith, and David D. Cheng. “Timing analysis of computer hardware”. In: *IBM Journal of Research and Development* 26.1 (1982), pp. 100–105 (cit. on p. 192).

[Huf52] David A. Huffman. “A method for the construction of minimum-redundancy codes”. In: *Proceedings of the Institute of Radio Engineers* 40.9 (1952), pp. 1098–1101 (cit. on pp. 31, 69, 119, 161, 162, 173–175).

[KBS98] Yuji Kukimoto, Robert K. Brayton, and Prashant Sawkar. “Delay-optimal technology mapping by DAG covering”. In: *Proceedings of the 35th Design Automation Conference*. 1998, pp. 348–351 (cit. on p. 196).

[KC66] T. I. Kirkpatrick and N. R. Clark. “PERT as an aid to logic design”. In: *IBM Journal of Research and Development* 10.2 (1966), pp. 135–141 (cit. on p. 192).

[Kee+11] Matthew Keeter, David Money Harris, Andrew Macrae, Rebecca Glick, Madeleine Ong, and Justin Schauer. “Implementation of 32-bit Ling and Jackson adders”. In: *Conference Record of the Forty Fifth Asilomar Conference on Signals, Systems and Computers*. 2011, pp. 170–175 (cit. on p. 55).

[Keu87] Kurt Keutzer. “DAGON: Technology binding and local optimization by DAG matching”. In: *Proceedings of the 24th Design Automation Conference*. 1987, pp. 341–347 (cit. on p. 196).

[Khr07] V. M. Khrapchenko. “On possibility of refining bounds for the delay of a parallel adder”. In: *Diskretnyi Analiz I Issledovanie Operatsii* 14.1 (2007), pp. 87–93. English translation: V. M. Khrapchenko. “On possibility of refining bounds for the delay of a parallel adder”. In: *Journal of Applied and Industrial Mathematics* 2.2 (2008), pp. 211–214 (cit. on p. 43).

[Khr67] V. M. Khrapchenko. “Asymptotic estimation of the addition time of a parallel adder”. In: *Problemy Kibernetiki* 19 (1967), pp. 107–122. English translation: V. M. Khrapchenko. “Asymptotic estimation of the addition time of a parallel adder”. In: *Systems Theory Research* 19 (1970), pp. 105–122 (cit. on pp. 56, 211, 217).

[Kno99] Simon Knowles. “A family of adders”. In: *Proceedings of the 14th IEEE Symposium on Computer Arithmetic*. IEEE. 1999, pp. 30–34 (cit. on pp. 34, 52).

[KR89] Kurt Keutzer and Dana Richards. “Computational complexity of logic synthesis and optimization”. In: *Proceedings of the International Workshop on Logic Synthesis*. 1989 (cit. on pp. 196, 201).

[Kra49] Leon G. Kraft, Jr. “A Device for Quantizing, Grouping, and Coding Amplitude-Modulated Pulses”. PhD thesis. Massachusetts Institute of Technology, 1949 (cit. on pp. 9, 30, 32).

[KRV07] Bernhard Korte, Dieter Rautenbach, and Jens Vygen. “BonnTools: Mathematical innovation for layout and timing closure of systems on a chip”. In: *Proceedings of the IEEE* 95.3 (2007), pp. 555–572 (cit. on p. 191).

[KS73] Peter M. Kogge and Harold S. Stone. “A parallel algorithm for the efficient solution of a general class of recurrence equations”. In: *Transactions on Computers* 100.8 (1973), pp. 786–793 (cit. on p. 54).

[KV18] Bernhard Korte and Jens Vygen. *Combinatorial Optimization: Theory and Algorithms*. 6th ed. 2018 (cit. on p. 22).

[KW90] Mauricio Karchmer and Avi Wigderson. “Monotone circuits for connectivity require super-logarithmic depth”. In: *SIAM Journal on Discrete Mathematics* 3.2 (1990), pp. 255–265 (cit. on p. 144).

[Lee76] J. van Leeuwen. “On the construction of Huffman trees.” In: *International Colloquium on Automata, Languages and Programming.* 3. 1976, pp. 382–410 (cit. on pp. 31, 32, 34, 80, 82).

[LF80] Richard E. Ladner and Michael J. Fischer. “Parallel prefix computation”. In: *Journal of the ACM* 27.4 (1980), pp. 831–838 (cit. on pp. 54, 55, 208, 209, 223–225).

[Liu+03] Jianhua Liu, Shuo Zhou, Haikun Zhu, and Chung-Kuan Cheng. “An algorithmic approach for generic parallel adders”. In: *Proceedings of the 2003 International Conference on Computer-Aided Design.* 2003, pp. 734–740 (cit. on p. 56).

[McC56] E. J. McCluskey, Jr. “Minimization of Boolean functions”. In: *The Bell System Technical Journal* 35.6 (1956), pp. 1417–1444 (cit. on p. 195).

[McG+93] Patrick McGeer, Jagesh Sanghavi, Robert Brayton, and Alberto Sangiovanni-Vicentelli. “Espresso-Signature: A new exact minimizer for logic functions”. In: *Transactions on Very Large Scale Integration (VLSI) Systems* 1.4 (1993), pp. 432–440 (cit. on p. 195).

[MD93] Frédéric Mailhot and Giovanni Di Micheli. “Algorithms for technology mapping based on binary decision diagrams and on Boolean operations”. In: *Transactions on Computer-Aided Design of Integrated Circuits and Systems* 12.5 (1993), pp. 599–620 (cit. on p. 196).

[Mer93] Jean P. Mermet. *Fundamentals and Standards in Hardware Description Languages.* 1993 (cit. on p. 195).

[MHK15] Igor L. Markov, Jin Hu, and Myung-Chul Kim. “Progress and challenges in VLSI placement research”. In: *Proceedings of the IEEE* 103.11 (2015), pp. 1985–2003 (cit. on p. 195).

[Mis+11] Alan Mishchenko, Robert Brayton, Stephen Jang, and Victor Kravets. “Delay optimization using SOP balancing”. In: *Proceedings of the 2011 International Conference on Computer-Aided Design.* 2011, pp. 375–382 (cit. on p. 196).

[MRV11] Dirk Müller, Klaus Radke, and Jens Vygen. “Faster min–max resource sharing in theory and practice”. In: *Mathematical Programming Computation* 3.1 (2011), pp. 1–35 (cit. on p. 202).

[Mur91] Saburo Muroga. “Computer-aided logic synthesis for VLSI chips”. In: *Advances in Computers.* Vol. 32. 1991, pp. 1–103 (cit. on p. 196).

[NC07] Gi-Joon Nam and Jason Cong. *Modern Circuit Placement: Best Practices and Results.* 2007 (cit. on p. 195).

[Ofm62] Yu P. Ofman. “On the algorithmic complexity of discrete functions”. In: *Doklady Akademii Nauk.* Vol. 145. 1. Russian Academy of Sciences. 1962, pp. 48–51. English translation: Yu P. Ofman. “On the algorithmic complexity of discrete functions”. In: *Soviet Physics Doklady.* Vol. 7. 1963, pp. 589–591 (cit. on pp. 54, 211, 217).

[Okl94] Vojin G. Oklobdzija. “Design and analysis of fast carry-propagate adder under non-equal input signal arrival profile”. In: *Conference Record of The Twenty-Eighth Asilomar Conference on Signals, Systems and Computers*. Vol. 2. 1994, pp. 1398–1401 (cit. on p. 56).

[Ott98] Ralph H. J. M. Otten. “Global wires harmful?” In: *Proceedings of the 1998 International Symposium on Physical Design*. 1998, pp. 104–109 (cit. on pp. 193, 200).

[PMB08] Stephen M. Plaza, Igor L. Markov, and Valeria Bertacco. “Optimizing non-monotonic interconnect using functional simulation and logic restructuring”. In: *Transactions on Computer-Aided Design of Integrated Circuits and Systems* 27.12 (2008), pp. 2107–2119 (cit. on p. 197).

[Qui52] W. V. Quine. “The problem of simplifying truth functions”. In: *The American Mathematical Monthly* 59.8 (1952), pp. 521–531 (cit. on p. 195).

[Roy+14] Subhendu Roy, Mihir Choudhury, Ruchir Puri, and David Z. Pan. “Towards optimal performance-area trade-off in adders by synthesis of parallel prefix structures”. In: *Transactions on Computer-Aided Design of Integrated Circuits and Systems* 33.10 (2014), pp. 1517–1530 (cit. on p. 55).

[Roy+15] Subhendu Roy, Mihir Choudhury, Ruchir Puri, and David Z. Pan. “Polynomial time algorithm for area and power efficient adder synthesis in high-performance designs”. In: *Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35.5 (2015), pp. 820–831 (cit. on p. 55).

[RP94] Curtis L. Ratzlaff and Lawrence T. Pillage. “RICE: Rapid interconnect circuit evaluation using AWE”. In: *Transactions on Computer-Aided Design of Integrated Circuits and Systems* 13.6 (1994), pp. 763–776 (cit. on pp. 193, 201, 202).

[RS87] Richard L. Rudell and Alberto Sangiovanni-Vincentelli. “Multiple-valued minimization for PLA optimization”. In: *Transactions on Computer-Aided Design of Integrated Circuits and Systems* 6.5 (1987), pp. 727–750 (cit. on p. 195).

[RSW03] Dieter Rautenbach, Christian Szegedy, and Jürgen Werber. *Asymptotically optimal Boolean circuits for functions of the form* $g_{n-1}(g_{n-2}(...g_3(g_2(g_1(x_1, x_2), x_3), x_4)..., x_{n-1}), x_n)$ *given input arrival times*. Technical Report 03931. Research Institute for Discrete Mathematics, University of Bonn, 2003 (cit. on pp. 58, 60).

[RSW06] Dieter Rautenbach, Christian Szegedy, and Jürgen Werber. “Delay optimization of linear depth Boolean circuits with prescribed input arrival times”. In: *Journal of Discrete Algorithms* 4.4 (2006), pp. 526–537 (cit. on pp. 10, 58–60, 159, 165, 177–182, 184, 185, 187, 197).

[RSW07] Dieter Rautenbach, Christian Szegedy, and Jürgen Werber. “The delay of circuits whose inputs have specified arrival times”. In: *Discrete Applied Mathematics* 155.10 (2007), pp. 1233–1243 (cit. on p. 55).

[Sap04] Sachin Sapatnekar. *Timing*. 2004 (cit. on p. 195).

[Sav98] John E. Savage. *Models of Computation*. Vol. 136. 1998 (cit. on p. 13).

[Sch15] Ulrike Elisabeth Schorr. “Algorithms for Circuit Sizing in VLSI Design”. PhD thesis. University of Bonn, 2015 (cit. on p. 195).

[Skl60] J. Sklansky. “Conditional-sum addition logic”. In: *IRE Transactions on Electronic computers* 2 (1960), pp. 226–231 (cit. on p. 54).

[Slo] Neil J. A. Sloane. *The On-Line Encyclopedia of Integer Sequences.* `https://oeis.org`. [Online; accessed 02-05-2020] (cit. on p. 149).

[SO96a] Paul F. Stelling and Vojin G. Oklobdzija. “Design strategies for the final adder in a parallel multiplier”. In: *Conference Record of The Twenty-Ninth Asilomar Conference on Signals, Systems and Computers.* 1996 (cit. on p. 56).

[SO96b] Paul. F. Stelling and Vojin G. Oklobdzija. “Design strategies for optimal hybrid final adders in a parallel multiplier”. In: *Journal of VLSI Signal Processing Systems for Signal, Image and Video Technology* 14.3 (1996), pp. 321–331 (cit. on p. 56).

[Spi14] Sophie Theresa Spirkl. “Boolean Circuit Optimization”. MA thesis. University of Bonn, 2014 (cit. on pp. 10, 56, 58, 60, 103).

[Sto+96] L. Stok, D. S. Kung, D. Brand, A. D. Drumm, A. J. Sullivan, L. N. Reddy, N. Hieter, D. J. Geiger, H. Chao, and P. J. Osler. “BooleDozer: Logic synthesis for ASICs”. In: *IBM Journal of Research and Development* 40.4 (1996), pp. 407–430 (cit. on p. 196).

[Syn20] Synopsys. *Synopsys Design Compiler.* `https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test.html`. [Online; accessed 02-May-2020]. 2020 (cit. on p. 196).

[Tei13] Marvin Tom Tasso Teichmann. “Timing-Optimale Platzierung im VLSI-Design”. German. BA thesis. University of Bonn, 2013 (cit. on p. 200).

[Tre+04] L. Trevillyan, D. Kung, R. Puri, L. Reddy, and M. Kazda. “An integrated environment for technology closure of deep-submicron IC designs”. In: *Design & Test of Computers* 21 (2004), pp. 14–22 (cit. on p. 196).

[UVS06] Christopher Umans, Tiziano Villa, and Alberto L. Sangiovanni-Vincentelli. “Complexity of two-level logic minimization”. In: *Transactions on Computer-Aided Design of Integrated Circuits and Systems* 25.7 (2006), pp. 1230–1246 (cit. on p. 195).

[VKT02] Navin Vemuri, Priyank Kalla, and Russell Tessier. “BDD-based logic synthesis for LUT-based FPGAs”. In: *Transactions on Design Automation of Electronic Systems* 7.4 (2002), pp. 501–525 (cit. on p. 204).

[Weg87] Ingo Wegener. *The Complexity of Boolean Functions.* 1987 (cit. on p. 55).

[Wei+12] Yaoguang Wei, Cliff Sze, Natarajan Viswanathan, Zhuo Li, Charles J. Alpert, Lakshmi Reddy, Andrew D. Huber, Gustavo E. Tellez, Douglas Keller, and Sachin S. Sapatnekar. “GLARE: Global and local wiring aware routability evaluation”. In: *Proceedings of the 49th Design Automation Conference.* 2012, pp. 768–773 (cit. on p. 202).

[Wer07] Jürgen Werber. “Logic Restructuring for Timing Optimization in VLSI Design”. PhD thesis. University of Bonn, 2007 (cit. on pp. 30, 32, 191).

[Win65] S. Winograd. “On the time required to perform addition”. In: *Journal of the ACM* 12.2 (1965), pp. 277–285 (cit. on p. 30).

[WRS07] Jürgen Werber, Dieter Rautenbach, and Christian Szegedy. “Timing optimization by restructuring long combinatorial paths”. In: *Proceedings of the 2007 International Conference on Computer-Aided Design*. 2007, pp. 536–543 (cit. on pp. 191, 197).

[WS58] A. Weinberger and J. L. Smith. “A logic for high-speed addition”. In: *National Bureau of Standards Circular* 591 (1958), pp. 3–12 (cit. on p. 34).

[YJ03] Wen-Chang Yeh and Chein-Wei Jen. “Generalized earliest-first fast addition algorithm”. In: *Transactions on Computers* 52.10 (2003), pp. 1233–1242 (cit. on p. 56).

[Zim98] Reto Zimmermann. “Binary Adder Architectures for Cell-Based VLSI and their Synthesis”. PhD thesis. Swiss Federal Institute of Technology in Zurich, 1998 (cit. on p. 54).