Near-Optimum Circuits for Binary Addition and Multi-Output And Functions

Dissertation

zur

Erlangung des Doktorgrades (Dr. rer. nat.)

der

Mathematisch-Naturwissenschaftlichen Fakultät

der

Rheinischen Friedrich-Wilhelms-Universität Bonn

Vorgelegt von

Susanne Armbruster

aus

München

Bonn, 03. Februar 2026

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

Gutachter/Betreuer:	Herr Professor Dr. Stephan Held
Gutachter:	Herr Professor Dr. Jens Vygen
Tag der Promotion:	17. April 2026
Erscheinungsjahr:	2026

Acknowledgments

This thesis is the result of many years of work during which I was supported by many people.

First and foremost, my greatest thank goes to my advisor Prof. Dr. Stephan Held for his guidance, support and his contributed ideas as well as the detailed feedback to my work. Additionally, I want to thank Prof. Dr. Jens Vygen for contributing ideas during various meetings. The entire professional staff provided excellent working conditions at the Institute for Discrete Mathematics. I had a great time there.

I want to thank Anna Silvanus for a warm welcome at the institute when I was a student, her close supervision and the work she put in to leave the understandable and well-working project BonnLogic. My joy for this project arose due to the interesting problems and the great people I was working with. I want to thank my longtime office partner Fine Foos for many discussions, Ulrich Brenner for working with me on And-Or paths; and Martin Nägele and Matthias Mnich for our common work on the Ordered Traveling Salesperson Problem (which unfortunately does not fit to the topic of this thesis).

Thanks also go to all the other colleagues I was working with: to the students Yannik Splitzley and Roxana Mittelberg for contributing to the BonnLogic project; to the IBM employees around Alex Suess for the great collaboration; to Luise Puhlmann, Niklas Schlomberg and some more for our teamwork on polynomial running time; and to all the other colleagues for a great time and interesting discussions.

Moreover, I want to thank Ulrich Brenner, Stephan Held, Adrian Riekert and Niklas Schlomberg for proofreading parts of this thesis.

Last but not least, I want to greatly thank my family: I am really grateful to my parents Ingrid and Hannes Armbruster for supporting me throughout my studies, and for their warm, objective and always helpful advice for my life; and to my brother and sister for making my life more lively. My deepest thanks go to my husband Adrian Riekert for the support, the suggestions, the patience, the appreciation and the love he has given me over the last ten years.

Contents

Chapter 1

Introduction

The design of fast and powerful computer chips is a challenging problem in the modern world whose importance is ever-growing. Chips are supposed to become both faster and smaller without increasing the power consumption too much, yielding a new challenge in every chip generation. Building such a chip from tiny transistors requires a long planning phase during which the types, amount, placement and connections of these transistors are computed. This phase is called the design process. While a lot of this work used to be done manually on early chips, up-to-date chips require complex tools to optimize the chip's properties, each solving different subtasks. Thus, chip design gives rise to a large number of complex and fascinating problems in combinatorial optimization that are both interesting as a standalone problem and very applied.

In this thesis, we focus on some problems arising within the subtask of logic optimization. Each chip consists of millions of basic building blocks called gates that, given a set of Boolean input variables, compute basic Boolean functions like NANDs or NORs. Gates consist of a small number of transistors and are interconnected with wires, sending the computed signal to other gates or to the outside of the chip. The directed graph-like structure formed by these gates is called a Boolean circuit. It models the physical implementation of Boolean functions by introducing vertices for every gate and by adding an edge between two gates whenever a signal gets sent between them. An example for such a circuit can be seen in Figure 1.1.

During the design process, the structure of the Boolean circuit is not fixed, but it can be optimized with respect to various objectives. Improving this circuit is the task of logic optimization and the topic of this thesis. Different parts of the chip are optimized with respect to different objectives and therefore using different algorithms. To ensure that the chip is sufficiently fast in the end, it is essential that the circuit has a small depth because every gate takes a certain delay until the result is computed und thus, long chains of gates lead to large delays. If the input signals are not available at the same time, this objective can be generalized to an arrival time model, in which each input comes with a given input arrival time. The arrival time of a gate is then given by the sum of the gate delay and the maximum predecessor arrival time. More general models even incorporate an edge delay for edges between gates or different pin delays for the different predecessors of a gate into the delay model.

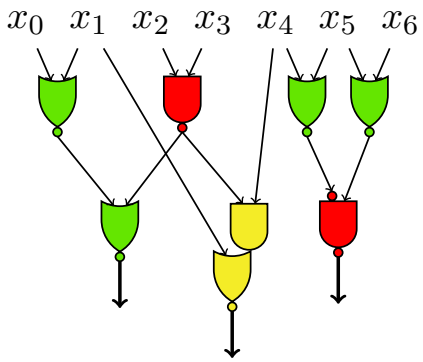


Figure 1.1: Example of a circuit on a chip. The inputs of the chip that each represent a Boolean variable are shown at the top as $x_0, \ldots, x_6$. The gates are the colored vertices; the red gates are NAND gates, the green gates are NOR gates and the yellow gate is an AOI gate. Throughout this thesis, they can be always recongnized by their form. Small dots represent an inversion. The arrows represent the signal flow and the thick arrows represent chip outputs.

Building fast Boolean circuits will usually be at the expense of the number of gates. More gates, i.e., an increased size of the circuit does not only lead to a larger chip but also to a larger power consumption and thus heat generation. Furthermore, more gates are more difficult to route, too. On timing-uncritical parts of a chip it is therefore essential to reduce the number of used gates as much as possible.

As optimizing Boolean formulae in general is a Σ_2^p complete problem as shown by Buchfuhrer and Umans [BU11], we cannot hope to tackle the problem as a whole but instead divide the problem into smaller pieces. We consider different kinds of Boolean functions that can be optimized more efficiently and restructure them using a small set of gate types. Afterward, we apply technology mapping to make use of the large library of gates that are available on modern chips.

First, we consider AND-OR paths, i.e., circuits that given some input variables $t_0, \ldots, t_{m-1}$ realize Boolean functions

$$t_0 \wedge \Big(t_1 \vee \big(t_2 \wedge (t_3 \vee \ldots t_{m-1})\big)\Big) \ .$$

Such functions arise in particular during the process of adding two binary encoded numbers. However, also many other functions can be broken down to computing AND-OR paths. After introducing some basics and some notation used for Boolean circuit optimization in Chapter 2, we study the problem of building fast AND-OR paths in Chapter 3. In joint work with Ulrich Brenner, we improve on the currently best known upper bound on the depth of an optimum restructuring of a given AND-OR path using a recursive algorithm similar to the approaches of Brenner and Silvanus [BS24]. By refining the analysis, we are able to present an algorithm computing a class of circuits of depth

$$\log_2 m + \log_2 \log_2 m$$

that asymptotically approaches

$$\log_2 m + \log_2 \log_2 m - 1 \ .$$

Moreover, we develop a lower bound on the rate of convergence. In addition, we show a modification for the algorithm such that it can be implemented using a linear number of gates by reusing gates that compute an AND or OR function on a subset of the inputs. We provide a set of AND and OR gates that can be reused every time we need an AND or an OR function and give a detailed analysis of the gates needed in addition to the reused gates. To derive a small linear size, we automate the procedure of finding small AND-OR path circuits for small numbers of inputs. Since our size analysis is based on a recursive strategy, this leads to significantly better size bounds on larger AND-OR paths, too. After this analysis, we show how to use the AND-OR paths we have built to derive adder circuits with a depth of at most

$$\log_2 m + \log_2 \log_2 m + \log_2 \log_2 \log_2 m + 5.6$$

while maintaining a linear size of at most $19n$.

In Chapter 4 we study the minimization of AND functions by themselves in a more general form. We consider a family of functions, each computing an AND on a subset of a given set of inputs. Such families of functions can possibly share some of the gates used to compute them. We consider the problem of finding a common circuit for all functions in the family with minimum size, i.e., with the minimum number of AND gates. We prove that it is APX-hard even in the case of every function only depending on three inputs by reducing vertex cover to this problem. Then we study approximation algorithms for instances with inputs per tree, that is, with at most three or at most four inputs per AND function. In these cases, the smallest overall circuit and the smallest depth-optimum circuit either coincide or can be reduced to each other while only losing a small factor in the approximation guarantee. For the problem of size minimization, we present a 1.212- and a 1.731-approximation for instances with at most three and at most four input variables per AND function, respectively. To achieve these results, we model the problem by a linear program. For small numbers of inputs per AND function, this linear program can be solved efficiently and the fractional solution can be rounded to an integer one. Afterward, we present an approach for the general problem working for an arbitrary but constant number of inputs. Assuming that k is the maximum number of inputs of an AND function, we give a $\frac{2}{3}k$-approximation. This is achieved by introducing a simple Greedy algorithm and establishing a lower bound on the optimum solution in cases where the Greedy algorithm does not perform well.

In Chapter 5, we show a practical approach to improve instances equivalent to the ones studied in Chapter 4. The practical implementation is incorporated into the logic restructuring tool BONNLOGIC which is part of the BONNTOOLS. We describe the setting as well as the previous approach to improve single-output AND functions. To apply our model, we collect a set of maximal AND functions and group them into instances, depending on their pairwise overlap in the input variables. Every instance is modeled as an integer linear program corresponding to the linear program of Chapter 4. Using an exact integer linear programming solver, we decrease the number of 2-ary gates contained in the instance, usually reaching the minimum. The derived new solutions are then mapped back to the chip and slightly improved for comparability. We present experimental results and different statistics on instances extracted from the EPFL benchmarks introduced by Amarú, Gaillardon, and De Micheli [AGD15] as well as on real world instances provided by our cooperation partner IBM. Additionally, we give an outlook of how to extend this integer linear program to arrival times.

Chapter 2

Preliminaries

In this chapter we introduce preliminaries including definitions and optimization directions that we study throughout this thesis. However, we stick to the basics needed throughout the thesis. More specific details that are only needed within a single chapter will be introduced at its beginning.

2.1 Boolean functions, formulae and circuits

Our notion of Boolean functions is based on Savage [Sav98] and Crama and Hammer [CH11]. We stick closely to Hermann [Her20].

2.1.1 Boolean functions

Definition 2.1.1. A **Boolean variable** is a binary variable taking values in $\{0,1\}$. We refer to the value of 1 as true and the value of 0 as false.

Given $n, m \in \mathbb{N}$, a **Boolean function on n Boolean variables**, which we also refer to as inputs, is a function $f\colon \{0,1\}^n \to \{0,1\}^m$.

We will often look at the Boolean input variables as a vector $x = (x_0, \dots, x_{n-1})$. In Chapter 3, we mostly study one-dimensional functions, i.e., $m = 1$.

Definition 2.1.2. Given a Boolean function $f\colon \{0,1\}^n \to \{0,1\}$ we call n its **arity**. A function of arity n is called ***n*-ary**.

In this thesis, we want to study different Boolean function families and their implementations as Boolean circuits. Basic building blocks for these will be the following functions.

Definition 2.1.3. We define the following operators on Boolean variables.

(i) The **And operator** $\cdot \wedge \cdot\colon \{0,1\} \times \{0,1\} \to \{0,1\}$ is defined by

$$x \wedge y \coloneqq \begin{cases} 1 & \text{if } x = y = 1 \ ; \\ 0 & \text{else } . \end{cases}$$

The And operator is also called a Boolean conjunction. More generally, we define the And operator $\bigwedge\colon \{0,1\}^n \to \{0,1\}$ on n Boolean variables to be

$$\bigwedge_{i=0}^{n-1} x_i \coloneqq x_0 \wedge x_1 \wedge \cdots \wedge x_{n-1} \ .$$

(ii) Analogously, the **OR operator** $\cdot \vee \cdot \colon \{0,1\} \times \{0,1\} \to \{0,1\}$ is defined by

$$x \vee y \coloneqq \begin{cases} 0 & \text{if } x = y = 0 \ ; \\ 1 & \text{else } . \end{cases}$$

The OR operator is also called a Boolean disjunction. More generally, we define the OR operator $\bigvee \colon \{0,1\}^n \to \{0,1\}$ on n Boolean variables to be

$$\bigvee_{i=0}^{n-1} x_i \coloneqq x_0 \vee x_1 \vee \cdots \vee x_{n-1} \ .$$

(iii) Moreover, the **XOR operator** $\cdot \oplus \cdot \colon \{0,1\} \times \{0,1\} \to \{0,1\}$ is defined by

$$x \oplus y \coloneqq x + y \pmod 2 \ .$$

More generally, we define the XOR operator $\bigoplus \colon \{0,1\}^n \to \{0,1\}$ on n Boolean variables to be

$$\bigoplus_{i=0}^{n-1} x_i \coloneqq x_0 \oplus x_1 \oplus \ldots \oplus x_{n-1} \ .$$

(iv) Last, the **NOT operator** $\overline{\cdot} \colon \{0,1\} \to \{0,1\}$ is given by

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

We call this also inversion or Boolean negation.

2.1.2 Boolean formulae

To compute the functions we want to study, we consider different realizations. A Boolean function can either be realized as a Boolean formula or as a Boolean circuit.

Definition 2.1.4. The set of **Boolean formulae** over input variables $x_0, \ldots, x_{n-1}$ consists of

- the constants $0, 1$ and the variables $x_0, \ldots, x_{n-1}$; and
- every formula ρ that arises from two other Boolean formulae ϕ, ψ as $\rho = \overline{\phi}$, $\rho = \phi \vee \psi$ or $\rho = \phi \wedge \psi$.

A formula ρ' is called **equivalent** to a formula ρ if it realizes the same Boolean function, i.e., $\rho(x) = \rho'(x)$ for all $x \in \{0,1\}^n$.

Every Boolean formula is realized by a Boolean function but Boolean functions have many different realizations as Boolean formulae. Example 2.1.5 shows three possible realizations of an XOR.

Example 2.1.5. The XOR function $f(x_0, x_1) = x_0 \oplus x_1$ given in Item (iii) of Definition 2.1.3 can for example be realized by the following three formulae.

$$\begin{aligned} \rho_1(x_0, x_1) &= (x_0 \wedge \overline{x_1}) \vee (\overline{x_0} \wedge x_1) \\ \rho_2(x_0, x_1) &= (x_0 \vee x_1) \wedge (\overline{x_0 \vee x_1}) \\ \rho_3(x_0, x_1) &= \overline{(x_0 \wedge x_1) \vee (\overline{x_0} \wedge \overline{x_1})} \end{aligned}$$

To transform formulae to equivalent formulae, we use the following basic laws.

Observation 2.1.6. Boolean formulae satisfy the following laws of computation.

(i) Commutativity: $x \wedge y = y \wedge x$ and $x \vee y = y \vee x$.

(ii) Associativity: $(x \wedge y) \wedge z = x \wedge (y \wedge z)$ and $(x \vee y) \vee z = x \vee (y \vee z)$.

(iii) Distributivity: $(x \wedge (y \vee z)) = (x \wedge y) \vee (x \wedge z)$ and $(x \vee (y \wedge z)) = (x \vee y) \wedge (x \vee z)$.

(iv) De Morgan law: $\overline{x \wedge y} = \overline{x} \vee \overline{y}$ and $\overline{x \vee y} = \overline{x} \wedge \overline{y}$.

Definition 2.1.7. The **dual Boolean formula** ρ^* for a formula ρ arises from ρ by changing every And operation to an Or operation and vice versa. The function computed by the dual formula is called the **dual function**.

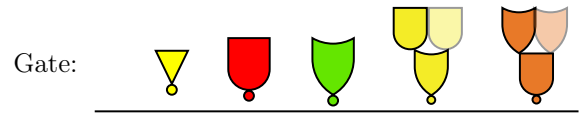
The dual formula is indeed dual in the sense that there is one-to-one correspondence between equivalent formulae to ρ and equivalent formulae to ρ^*. Thus, we can use in Chapter 3 that a formula for the primal function directly implies a formula of the same size, i.e., the same number of operations for the dual function.

2.1.3 Boolean circuits

In this thesis, we are given a certain set of gates called basis and aim to build good circuits based on this set of gates.

Definition 2.1.8. A **basis** is a set Ω of Boolean functions, possibly of different arities. Each element $\phi \in \Omega$ is called a gate type.

A basis typically contains sufficiently many gate types to ensure that every possible Boolean function can be computed using only gates of the basis. A minimal such basis is for example an And and an inverter. An example of a typical basis provided for chips can be seen in Figure 2.1. We use a basis to construct a circuit defined as follows.



Name:	Inv	Nand	Nor	Aoi	Oai
Arities:	1	2,3,4	2,3,4	3,4,5,6	3,4,5,6

Figure 2.1: Possible basis consisting of five different gate types. The dot at the bottom of the gates always indicates an inversion. The symbols with a round bottom shape are And operations whereas the shapes with the pointed bottom are Or operations.

Definition 2.1.9. A **circuit** over a basis Ω is a directed acyclic graph $G = (\mathcal{V}, \mathcal{E})$ together with a partition $\mathcal{V} = \mathcal{I} \cup \mathcal{G}$ into labeled inputs and labeled gate vertices, satisfying the following properties.

(i) Each vertex $v \in \mathcal{I}$ does not have any inputs, i.e. $\delta^-(v) = \emptyset$. It is labeled either with a Boolean variable x_v or with a constant 0 or 1.

(ii) Each vertex $v \in \mathcal{G}$ is called a gate vertex, and it is labeled with a gate $\phi \in \Omega$ of arity $\delta^-(v) \geq 1$. It comes along with a fixed ordering on the predecessors $\delta^-(v)$ of v.

(iii) There is a subset $\emptyset \neq \mathcal{O} \subseteq \mathcal{G}$ of vertices, which we call outputs. It contains every vertex $v \in \mathcal{V}$ for which $\delta^+(v) = \emptyset$.

Note that the fixed ordering on the predecessors for every vertex is important because some gates do not compute a result that is symmetric in the inputs. Thus, this fixed ordering indicates which input of the gate corresponds to which predecessor.

Notation 2.1.10. For every vertex $v \in \mathcal{V}$ we define $\mathcal{I}_v$ to be the set of inputs i for which there exists an i-v-path.

Definition 2.1.11. Let C be a circuit over a basis Ω. The **Boolean function f_v associated with $v \in \mathcal{V}$** is recursively given by

(i) the label of v if $v \in \mathcal{I}$ is an input (i.e., x_v or 0 or 1); or

(ii) $\phi_v(v_0, \ldots, v_k)$ if $v \in \mathcal{G}$ is a gate vertex and $v_0, \ldots, v_k$ is the fixed ordering on the predecessors of v.

We call the set of Boolean functions associated with the outputs the set of realized functions.

Definition 2.1.12. Two circuits are **equivalent** if they realize the same set of functions.

This notion is very important especially for Chapter 5 because our main motivation in this thesis is logic restructuring, that is to find an equivalent circuit for a given circuit from a chip. Thus, the Boolean functions are not to be changed at the outputs but just internally.

We call a circuit a **formula circuit** if every vertex $v \in \mathcal{V}$ has at most one successor. Single output formula circuits over the basis $\Omega = \{\text{AND}, \text{OR}, \text{NOT}\}$ and formulae for a given Boolean function have a one-to-one correspondence. On the one hand, every formula can be transformed into a formula circuit by using the decomposition implied by Definition 2.1.4. For every step of decomposition, a gate representing the used operator is inserted and the predecessor gates are then recursively determined by decomposing the two subformulae. On the other hand, the formula corresponding to a formula circuit can be determined by for every gate applying its operation onto the formulae recursively computed for the predecessors. This correspondence is consistent with equivalence: if two formula circuits are equivalent, then their corresponding formulae are also equivalent.

2.2 Optimization directions

We continue by introducing the most important metrics that we optimize throughout this thesis, and we defer the precise description of the problems including the description of the special circuits we optimize to the corresponding chapters. One of the most commonly studied metrics is the depth of the circuit.

Definition 2.2.1. Let C be a Boolean circuit. For a vertex $v \in \mathcal{V}(C)$ we set its **depth** $\text{depth}(v)$ to be the maximum number of edges of any directed path of C that ends in v. The depth of C is given by the maximum depth of any vertex $v \in C$.

Note that the depth of a circuit C is equal to the maximum number of edges on any directed path in C. On a chip, every vertex of the circuit is represented by a gate and every input represents a signal arriving on the chip. As these inputs often do not all arrive at the same time, the depth of a circuit is often generalized by introducing a delay model including arrival times for every vertex.

Definition 2.2.2. Let C be a Boolean circuit on inputs $x = (x_0, \dots, x_{n-1})$ over a basis Ω. Assume you are given gate delays $d\colon \Omega \to \mathbb{N}$. Additionally, let $a(x_0), \dots, a(x_{n-1}) \in \mathbb{N}$ be arrival times for inputs $x_0, \dots, x_{n-1}$. For any non-input vertex $v \in \mathcal{G}$ we set its **arrival time**

$$a(v) \coloneqq \max_{w \in \delta^-(v)} a(w) + d(\phi(v)).$$

The **delay of C** is given by $\operatorname{delay}(C) \coloneqq \max_{v \in \mathcal{G}(C)} a(v)$.

For most of this thesis, we deal with 2-ary gates only. As the gate delay of Nand and Nor gates are usually pretty similar, we stick to a simple delay model, in which every vertex causes an additional delay of 1. We call this the simple delay model. Using this delay model in the case of uniform input arrival times, i.e., $a(x) \equiv 0$, the depth of a vertex is precisely its arrival time. Thus, the maximum arrival time is indeed a generalization of the depth of a circuit.

As we discuss in Chapter 5, the actual delay is more complicated: there is not only a delay on the gates but also a delay on the edges, and the delay of a gate is not equal for all the predecessors. However, giving an exact description of the actual delay is basically impossible, so we stick to these models for now.

Apart from delay, a very important metric for chips is power consumption. This is well minimized by decreasing the number of gates a circuit contains, especially in the setting with 2-ary gates only, in which we will be in Chapter 3 and Chapter 4.

Definition 2.2.3. The **size** of a circuit is the number of gates of C.

Depth and size will be our main objectives for this work. The problem we would ideally want to solve is the following.

Circuit Delay Optimization Problem	
Instance:	A Boolean function $f\colon \{0,1\}^n \to \{0,1\}$ on inputs x, arrival times $a(x)$ and a basis Ω.
Task:	Compute a circuit over Ω realizing $f(x)$ with the minimum possible delay.

This problem is NP-hard even for uniform arrival times, which can be shown by a simple reduction to the Satisfiability problem (see Hermann [Her20] for example). We study this problem in the special case of And-Or path restructuring, i.e., of restructuring a path that consists of alternating And and Or gates (see Definition 3.1.1).

When optimizing size, one might instead want to solve the following problem.

Multi-Output Circuit Size Optimization Problem	
Instance:	A Boolean function $f\colon \{0,1\}^n \to \{0,1\}^m$ on inputs x and a basis Ω.
Task:	Compute a circuit over Ω realizing $f(x)$ with the minimum number of gates.

This problem is also NP-hard as shown by Ilango, Loff, and Oliveira [ILO20]. In Chapter 4 we study this problem in the special case of f being a family of AND operators. This is still APX-hard as we show, but we approach the problem using approximation algorithms.

In future work it might be worth to also consider the fanout of a vertex as this plays a role in the actual delay one might encounter on a chip.

Definition 2.2.4. Let C be a circuit. The **fanout** of a vertex $v \in \mathcal{V}(C)$ is the number of its successors. The **fanout** of C is the maximum fanout of any vertex in C.

We do not optimize the fanout in this thesis though.

Chapter 3

Depth optimization of And-Or paths and adders

This chapter is joint work with Ulrich Brenner. More details on the work distribution can be found in the contributions part at the end of the introduction (see page 25).

3.1 Introduction

In this chapter, our goal is to find fast and small circuits realizing a special kind of Boolean functions, the And-Or paths. As we will see later, they are very useful especially when it comes to building fast adder circuits.

Definition 3.1.1. Let $m \in \mathbb{Z}_{>0}$ and let $t = (t_0, \ldots, t_{m-1})$ be Boolean variables. The (primal) **And-Or path** on m inputs is recursively defined as

$$g(t) := \begin{cases} t_0 & m = 1 \\ t_0 \wedge g^*(t_1, \ldots, t_{m-1}) & m \geq 2 \end{cases}. \tag{3.1}$$

Its dual is defined by

$$g^*(t) = \begin{cases} t_0 & m = 1 \\ t_0 \vee g(t_1, \ldots, t_{m-1}) & m \geq 2 \end{cases}. \tag{3.2}$$

We call a circuit realizing an And-Or path an And-Or path circuit. Note that the function $g^*(t)$ is indeed the dual formula to $g(t)$ in the sense of Definition 2.1.7. Due to the duality principle, every circuit for the dual And-Or path can be turned into a circuit for the primal And-Or path by the precisely exchanging Ands and Ors. Thus, the task of finding a depth minimum primal And-Or path circuit is equivalent to finding a depth minimum dual And-Or path circuit.

An example of an And-Or path circuit is given in Figure 3.1. It is a straightforward realization of $g((t_0, \ldots, t_6))$ with depth 6 and size 6. Such a simple realization is called a **standard And-Or path circuit**. It is optimum in terms of size but very bad in terms of depth.

The problem we approach in this chapter is the following.

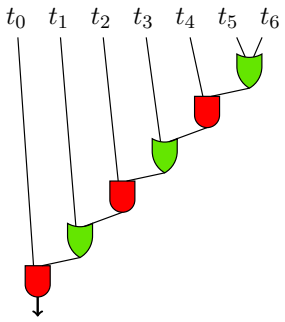


Figure 3.1: Standard AND-OR path circuit for $g((t_0, \ldots, t_6))$. The circuit has input variables $t_0, \ldots, t_6$. The AND gates are shown in red, the OR gates in green. Their inputs are determined by the edges adjacent to the upper part whereas the output is returned at the bottom. The output gate is marked by an arrow. We omit the arrows in between for simplicity, every edge is oriented downwards.

AND-OR PATH CIRCUIT OPTIMIZATION:
Instance: $m \in \mathbb{N}$ and Boolean input variables $t = (t_0, \ldots, t_{m-1})$.
Task: compute a circuit over $\Omega = \{\text{AND2}, \text{OR2}\}$ realizing $g(t)$ with the minimum possible depth.

Computing efficient AND-OR path circuits is a crucial and fundamental task in chip design since it has many important uses that we elaborate in the following.

3.1.1 Applications of AND-OR paths

Binary addition

The most important and very classical application for optimized AND-OR paths is the construction of fast adder circuits. Assume that we want to add two r-bit numbers $a = \sum_{i=0}^{r-1} a_i 2^i$ and $b = \sum_{i=0}^{r-1} b_i 2^i$, i.e., given $(a_0, \ldots, a_{r-1})$ and $(b_0, \ldots, b_{r-1})$, we want to compute bits $(s_0, \ldots, s_r)$ such that $\sum_{i=0}^{r} s_i 2^i = a + b$. This task can be reduced to the computation of the corresponding carry bits which are defined as follows.

Definition 3.1.2. Let $a = \sum_{i=0}^{r-1} a_i 2^i$ and $b = \sum_{i=0}^{r-1} b_i 2^i$ be two r-bit numbers. For $i \in \{0, \ldots, r-1\}$, we call $g_i = a_i \wedge b_i$ the i-th **generate signal** and $p_i = a_i \oplus b_i$ the i-the **propagate signal**. The **carry bits** are recursively given by $c_0 = 0$ and $c_{i+1} = g_i \vee (p_i \wedge c_i)$ for $0 \leq i \leq r-1$.

Figure 3.2 shows the carry bits in an example computation. When all carry bits are known, the sum bits s_i can be easily computed by setting $s_i = c_i \oplus p_i$ for $i \in \{0, \ldots, r-1\}$ and $s_r = c_r$. Conversely, the carry bits can be read directly from the input and the result, so computing the sum and the carry bits are in fact equivalent tasks. Consequently, any adder circuit developed so far explicitly computes the carry bits. Note that the carry bit c_{i+1} can be computed via the following AND-OR path

$$\begin{aligned} c_{i+1} &= g_i \vee (p_i \wedge c_i) = g_i \vee \Big(p_i \wedge \big(g_{i-1} \vee (p_{i-1} \wedge c_{i-1})\big)\Big) \\ &= g_i \vee \Big(p_i \wedge \Big(g_{i-1} \vee \big(p_{i-1} \wedge \big(g_{i-2} \vee \ldots (p_1 \wedge g_0)\big)\big)\Big)\Big) \ . \end{aligned}$$

$$\begin{array}{cccccccc} & & 1 & 1 & 0 & 1 & 1 \\ + & & 1 & 1 & 0 & 0 & 1 \\ & 1 & 1 & 0 & 1 & 1 & 0 \\ \hline & 1 & 1 & 0 & 1 & 0 & 0 \end{array}$$

Figure 3.2: Example for the carry bit computation when adding the binary encoded numbers $a = 27$ and $b = 25$. The carry bits are shown in blue. The result of the addition is shown below the line: the number 52 in binary encoding.

To derive an adder circuit, one needs to compute all the carry bits. We define the problem as follows.

Adder Circuit Optimization:	
Instance:	$n \in \mathbb{N}$ and n input variable pairs $(p_0, g_0, \ldots, p_{n-1}, g_{n-1})$.
Task:	compute a circuit over $\Omega = \{\text{And2}, \text{Or2}\}$ computing the carry bits $c_1, \ldots, c_n$.

Faster And-Or path circuits also allow for a faster computation of carry bits, which thus imply faster adder circuits. We analyze this step for our And-Or path circuits in Section 3.7.

Comparators

Additionally, And-Or paths can also be used to compute digital comparators. Assume that given the two r-bit numbers $a = \sum_{i=0}^{r-1} a_i 2^i$ and $b = \sum_{i=0}^{r-1} b_i 2^i$ one wants to compute the Boolean function $\text{gr}(a, b)$ which is 1 if a is greater than b. Let $d_i = a_i \wedge \neg b_i$ be true if and only if a_i is larger than b_i; and let $e_i = \neg a_i \oplus b_i$ be true if and only if $a_i = b_i$ for $i \in \{0, \ldots, r-1\}$. Then $\text{gr}(a, b)$ can be computed by evaluating the following And-Or path

$$\text{gr}(a, b) = d_{r-1} \vee \Big(e_{r-1} \wedge \big(d_{r-2} \vee \ldots (e_1 \wedge d_0) \ldots \big) \Big) \ .$$

Of course, one would get the same result by computing the difference between a and b and check for non-negativity. However, a customized circuit just for comparing numbers will be smaller and typically faster than an adder circuit that has to compute the whole sum and not just the last bit of it. As comparing numbers is a fundamental task, many authors optimized comparator circuits under practical aspects for given technologies, see e.g., Chual, Kumarl, and Sireesah [CKS17], Tyagi and Pandey [TP20], and Sunkireddy et al. [Sun+22]. Here, we focus on the theoretical basis of the optimization and follow a general technology-independent approach.

Note that modern CPU kernels contain hundreds of comparators and adders in their arithmetic logic units. Adders are not only used to add two integers in the code but the control logic, which forms the largest and most complex part of the CPU, also contains many incrementers that use adders internally. Thus, their efficiency is of great importance because they form crucial parts of the chip.

Subpaths of general circuits

Apart from these specific applications, And-Or path optimization can also be used for restructuring general logic circuits. To this end, timing-critical paths on a chip

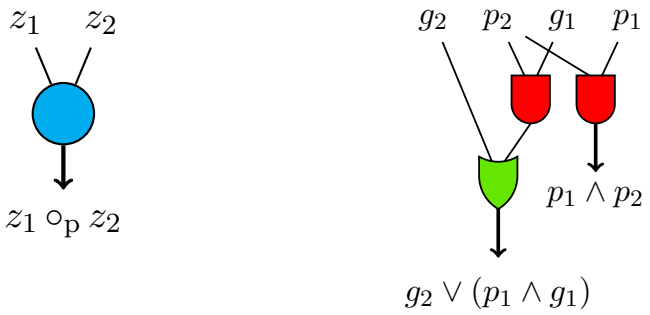
can be modeled by AND-OR paths (after moving all negations to the inputs and the outputs of the current part of the logic). Then, these AND-OR paths can be realized by efficient circuits and these circuits can be mapped back into the given technology. Details of this approach are described e.g. in Werber, Rautenbach, and Szegedy [WRS07] and Brenner and Silvanus [BS23] who successfully apply this method within BONNLOGIC which is part of the BONNTOOLS and more closely described in Section 5.2.

3.1.2 Previous work

Constructing fast adder circuits is a classical optimization problem that has been studied since the 1960s. Most previous approaches for fast AND-OR path circuits have been formulated in the context of adders.

The first approaches were based on prefix operators. The **adder prefix operator** is the binary operator $\circ_p \colon \{0,1\}^2 \times \{0,1\}^2 \to \{0,1\}^2$ that is given by

$$\begin{pmatrix} g_1 \\ p_1 \end{pmatrix} \circ_p \begin{pmatrix} g_0 \\ p_0 \end{pmatrix} = \begin{pmatrix} g_1 \vee (p_1 \wedge g_0) \\ p_1 \wedge p_0 \end{pmatrix} .$$



(a) A prefix gate in a prefix graph as defined below. It has 2 inputs and 1 output.

(b) Implementation of the adder prefix operator as a circuit over 2-ary gates.

Figure 3.3: This example shows the difference between a prefix gate in a prefix graph on the left and the implementation of the operator on the right.

Note that this operator is associative. Figure 3.3 shows an implementation of it on single-output gates. We call the gates arising from this operator **adder prefix gates**, and we define auxiliary variables $z_i = \binom{g_i}{p_i}$ for all $0 \leq i \leq m-1$. The prefix operator can be used to compute the carry bits as defined in Definition 3.1.2 by computing

$$\begin{aligned} \begin{pmatrix} c_{i+1} \\ p_i \wedge p_{i-1} \wedge \cdots \wedge p_0 \end{pmatrix} &= \begin{pmatrix} g_i \\ p_i \end{pmatrix} \circ_p \begin{pmatrix} g_{i-1} \\ p_{i-1} \end{pmatrix} \circ_p \cdots \circ_p \begin{pmatrix} g_0 \\ p_0 \end{pmatrix} \\ &= z_i \circ_p z_{i-1} \circ_p \cdots \circ_p z_0 \ , \end{aligned}$$

where g_i and p_i are the generate and propagate signals defined in Definition 3.1.2. Using this prefix operator, an adder circuit can be obtained from the m input pairs $(p_0, g_0), \ldots, (p_{m-1}, g_{m-1})$ by computing $z_i \circ_p z_{i-1} \circ_p \cdots \circ_p z_0$ for all $0 \leq i \leq m-1$. Hiding the internal implementation of this prefix operator, one can consider the general prefix graph problem.

Parallel Prefix Graph Problem
Instance: an associative operator $\circ$ and a set of inputs $z_0, \ldots, z_{m-1}$.
Task: construct a circuit computing $z_i \circ z_{i-1} \circ \cdots \circ z_0$ for all $0 \leq i \leq m-1$.

We call a solution to this problem a **prefix graph**. Every prefix graph corresponds to a solution for the adder problem by replacing every prefix gate with an implementation of the adder prefix operator. However, not every adder can be viewed as a prefix graph of course.

Note that the size of a prefix graph is multiplied by 3 when replacing the prefix operator by 2-ary gates. Also, the depth d increases during this replacement to a value between d and $2d$, depending on the precise prefix graph.

For the following overview, we assume that the number of inputs m is a power of 2. We start by giving an overview of the first approaches to build adders using prefix graphs.

Since the 1960s there have been various attempts to find a good tradeoff between depth, size and the maximum fanout of a prefix graph. The minimum possible depth is $\log_2 m$, and it can easily be reached. Sklansky [Skl60] showed this using a size of $\frac{1}{2}m \log_2 m$ and a maximum fanout of $\frac{1}{2}m + 1$. Ofman [Ofm62] gave the first logarithmic depth linear size adder with depth $\log_2 m$, size $3m - \log_2 m - 2$ and fanout $\frac{1}{2}\log_2 m$. Later, the famous adder by Kogge and Stone [KS73] achieved depth $\log_2 m$ with a size of $m \log_2 m - \frac{m}{2}$ while maintaining a fanout of 2. This adder is very often used in practice due to its small fanout and optimum prefix graph depth. Ladner and Fischer [LF80] give a flexible adder that, given a constant f, obtains an adder with depth $\log_2 m + f$, fanout $2^{-f-1}m + 1$ and size $2(1 + 2^{-f})m$. The size analysis was presented by Wegener [Weg87]. Brent and Kung [BK82] introduced the first linear size adder with fanout 2, achieving a depth of $2\log_2 m - 1$, however. Zimmermann [Zim98] gives an overview of the construction of these as well as further adders. Since all circuits mentioned above are prefix graphs, they have to be realized by gates computing only a single output when implemented in practice. Thus, all the above approaches only lead to a depth bound of at least $2\log_2 m$ and thrice the size in terms of And and Or gates.

To overcome this problem, there is further work which constructs adders by prefix graphs but analyzes their depth or arrival time in terms of And and Or gates. For this, assume the inputs come with given arrival times $a(g_i) = a(p_i)$, the gate delays are all one, and let $V = \sum_{i=0}^{m-1} 2^{a(g_i)}$. Note that for uniform arrival times $a \equiv 0$, we have $V = m$. Rautenbach, Szegedy, and Werber [RSW06] provide an And-Or path circuit with delay $1.44 \log_2 V + \mathcal{O}(1)$ with linear size. This can be transformed into an adder circuit with delay $2\log_2 V + 6\log_2 \log_2 m + \mathcal{O}(1)$ and at most $\mathcal{O}(m \log_2 \log_2 m)$ gates as shown by Rautenbach, Szegedy, and Werber [RSW07]. This has been improved upon by Held and Spirkl [HS17b]. They provide an adder with delay $1.441 \log_2 V + 5 \log_2 \log_2 m + 4.5$ and a size of $6m \log_2 \log_2 m$. Moreover, they give a lower bound on the minimum delay of an And and Or circuit constructed via prefix graphs of

$$\log_\phi \left(\sum_{i=0}^{m-1} \phi^{a(g_i)} \right) - 1 \geq 1.44 \log_2 \left(\sum_{i=0}^{m-1} \phi^{a(g_i)} \right) ,$$

where $\phi = \frac{1+\sqrt{5}}{2} \approx 1.618$ is the golden ratio. In the case of uniform arrival times, this corresponds to a lower bound of $1.44 \log_2 m - 1$, which implies that their prefix

graph algorithm is close to optimum in this case. However, it also implies that prefix graph adders are not the best way to construct a fast adder.

To construct even faster adders, one needs to optimize the depth directly with respect to AND and OR gates instead of constructing prefix graphs. Thereby, adders with depths of $(1 + o(1)) \log_2(m)$ could be achieved. Khrapchenko [Khr67] provides an adder of depth $\log_2 m + 7\sqrt{2 \log_2 m} + 14$ and size $9m$. Later, Held and Spirkl [HS17a] provide a circuit with depth $\log_2 m + o(\log_2 m)$, size $\mathcal{O}(m)$ and fanout 2.

The best guarantees for the depth of adders were achieved by an approach of recursive splitting of AND-OR paths. In the following, let m be the number of alternating inputs of the AND-OR path. We give an overview of the achieved depths for AND-OR paths. Grinchuk [Gri09] reaches an upper bound on the AND-OR path depth of $\log_2 m + \log_2 \log_2 m + 3$. Note that Grinchuk even claims an upper bound of $\log_2 m + \log_2 \log_2 m + 2$ but there is a gap in their analysis as pointed out by Ley [Ley22]. Their approach and analysis was further improved to $\log_2 m + \log_2 \log_2 m + 1.58$ by Hermann [Her20], to $\log_2 m + \log_2 \log_2 m + 1.5$ by Ley [Ley22] and to $\log_2 m + \log_2 \log_2 m + 0.65$ by Brenner and Silvanus [BS24]. All these circuits can be implemented with a linear size, the last ones having a size of at most $3.67m - 2$. One can construct a linear size adder from these AND-OR paths, too, but loses a bit in the depth. Brenner and Silvanus [BS24] achieve a depth of $\log_2 m + \log_2 \log_2 m + \log_2 \log_2 \log_2 m + 7.6$ and a size of at most $16.7m$. Note that all the mentioned adders can be built in polynomial time. We will follow their approach for AND-OR path restructuring and develop it further to achieve linear size AND-OR paths with even smaller depth and size. Moreover, we show how our AND-OR paths can be used in a linearization framework to create adder circuits that have a smaller depth than the ones mentioned above, too.

The upper bound of $\log_2 m + \log_2 \log_2 m + 0.65$ on the depth of an AND-OR path can only be improved by an additive constant since Commentz-Walter [Com79] showed that there is a constant c such that no AND-OR path circuit on m inputs consisting of AND and OR gates can have a depth of less than $\log_2 m + \log_2 \log_2 m + c$. Hitzschke [Hit18] showed that for large m, the constant c can be chosen arbitrarily close to -5. Later, Dietz [Die24] improved this asymptotic lower bound by 1 by showing that for any $\epsilon > 0$ for sufficiently large m, there is a lower bound of $\log_2 m + \log_2 \log_2 m - 4 - \epsilon$ on the depth of any AND-OR path circuit on m inputs consisting of AND and OR gates.

At last, note that even though the depth with respect to AND and OR gates is much more appropriate than measuring the depth with respect to prefix gates, every such circuit will in the end undergo a technology mapping as described in Section 5.5. This can change the depth or delay again. However, different gates have different delays depending on the library. This is the reason why we just like most other authors do not optimize the depth after technology mapping directly but instead focus on improving the depth of the circuit with respect to AND and OR gates. They still give a pretty good prediction of the delay after technology mapping anyway.

3.1.3 Our results

Our goal is to further reduce the gap between the depth of the best known AND-OR path circuits and the lower bound proven in Commentz-Walter [Com79] and Dietz [Die24]. As pointed out above, the currently best upper bound of $\log_2 m + \log_2 \log_2 m + 0.65$ due to Brenner and Silvanus [BS24] can only be improved by at most an additive constant. However, an improvement in the depth d by an additive

constant of 1 corresponds to almost doubling the number of inputs that can be added using a fixed depth circuit and is thus a significant step. We improve both the depth and the size in the following theorem.

Theorem 3.1.3. *Let $t = (t_0, \dots, t_{m-1})$ such that $m \geq 1$. There exists a polynomial time algorithm that computes a circuit C for the* And-Or *path $g(t)$ of depth*

$$\operatorname{depth}(C) \begin{cases} = \lceil \log_2 m \rceil & \text{for } m \leq 3 \text{ and} \\ \leq \log_2 m + \log_2 \log_2 m & \text{for } m \geq 4 \end{cases}$$

and size

$$\operatorname{size}(C) \leq 2.63m - 3 \ .$$

This theorem is a direct consequence of Theorem 3.4.6 proving the depth guarantee, and Corollary 3.6.30 proving the size guarantee. They will be proven in Section 3.4 and Section 3.6, respectively. We remark that this size can only be achieved by a computer-aided proof where we automatically compute the sizes of small circuits. However, apart from the computer-aided proof, we give a slightly weaker bound of $4.08m - 3$ that can be proven and verified manually. We give both proofs in Section 3.6.

Note that the bound of Theorem 3.1.3 is tight for $m = 4$, so in general it cannot be improved. However, for larger values of m, some better bound can be found as stated in the following theorem that follows from Theorem 3.5.1 that will be proven in Section 3.5.

Theorem 3.1.4. *Let $\varepsilon > 0$. There exists an $M \in \mathbb{Z}_{\geq 0}$ with the following property. For all $m \in \mathbb{Z}_{\geq M}$ and $t = (t_0, \dots, t_{m-1})$, there exists a polynomial time algorithm that computes a circuit C for the* And-Or *path $g(t)$ of depth*

$$\operatorname{depth}(C) \leq \log_2 m + \log_2 \log_2 m - (1 - \varepsilon)$$

and size

$$\operatorname{size}(C) \leq 2.63m - 3 \ .$$

Note that these bounds differ from the lower bounds given by Dietz [Die24] by only an additive term of approximately 3. Moreover, since circuit depths are integral, we in fact get an upper bound of $\lfloor \log_2 m + \log_2 \log_2 m - (1-\varepsilon) \rfloor$ on the depths. Thus, the gap to the lower bound is for infinitely many instances arbitrarily close to 2. In particular, there is no constant $D > 2$ such that the depth bound presented in Theorem 3.1.4 could be improved for any And-Or path circuit by an additive term of D. This leaves only a tiny possible room for improvement upon our depth bounds.

Figure 3.4 illustrates the development of the upper and lower bounds on the smallest additive term c, such that an And-Or path circuits on m inputs with depth $\log_2 m + \log_2 \log_2 m + c$ exists.

We use our results on And-Or paths to improve on the best known adders in the following theorem.

Theorem 3.1.5. *Let $n \in \mathbb{Z}_{\geq 4}$. There exists an algorithm to construct an adder circuit A_n^2 on n input pairs in running time $\mathcal{O}(n \log_2 n)$ with depth*

$$\operatorname{depth}(A_n^2) \leq \log_2 n + \log_2 \log_2 n + \log_2 \log_2 \log_2 n + 5.6$$

and size

$$\operatorname{size}(A_n^2) \leq 19n \ .$$

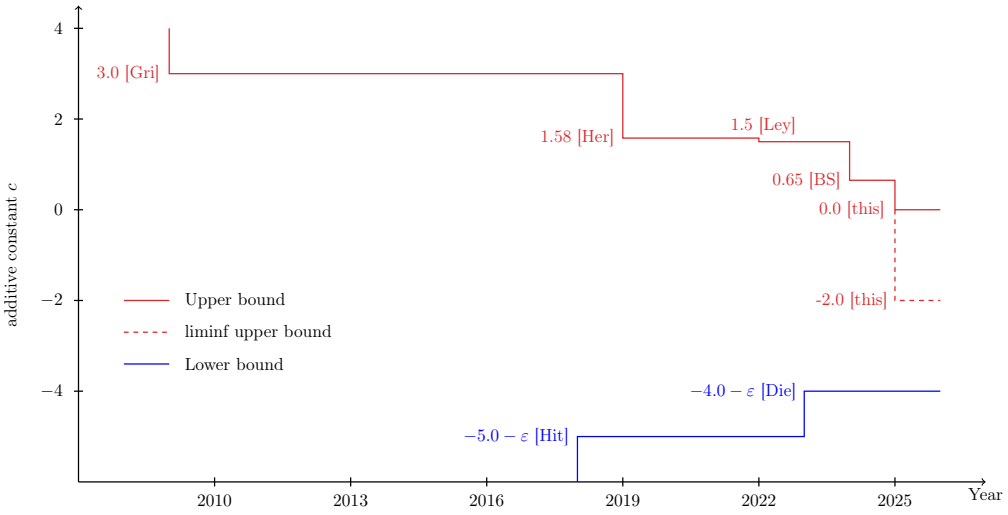


Figure 3.4: Summary of improvements upon the upper and lower bounds on the depth of an optimum AND-OR path. Development of the depth bounds over time. The given constant c is the constant such that the depth bound is $\log_2 m + \log_2 \log_2 m + c$. The red line shows the iteratively achieved upper bounds. The dotted line refers to the limes inferior of the value provided by Theorem 3.1.4. The blue line shows the improvement of the asymptic lower bound (i.e., the corresponding bound holds only for sufficiently large instances). In any case, even the limes inferior of the upper bounds cannot fall below the blue line.

Implications of our results for adders and comparators

As mentioned, the depth of classical adders like the Kogge-Stone adder introduced by Kogge and Stone [KS73] is usually given with respect to so-called prefix gates, and is in terms of AND and OR gates twice as large. Thus, the Kogge-Stone adder computes circuits with a depth that is about twice the depth of circuits resulting from our algorithm.

Additionally, comparing numbers using AND-OR paths is faster than the common approach of subtracting the numbers and checking the result for non-zero, unless we allow a super-linear size for the circuits. When comparing two numbers a and b as above, the Boolean variables $d_i = a_i \wedge \neg b_i$ and $e_i = \neg a_i \oplus b_i$ can be computed using a single gate. Thus, the total depth of comparators resulting from our AND-OR paths is $\lfloor \log_2 m + \log_2 \log_2 m + 1 \rfloor$ and can be reduced to $\lfloor \log_2 m + \log_2 \log_2 m + \varepsilon \rfloor$ asymptotically. Note that this result can be achieved with a linear size.

For adders, we have to compute all carry bits simultaneously, i.e., a linear size for computing only one carry bit would immediately lead to a quadratic size in total. When we restrict ourselves to adder circuits of linear total size, the best previous depth that is known in literature is $\log_2 m + \log_2 \log_2 m + \log_2 \log_2 \log_2 m + 6.6$ (see Brenner and Silvanus [BS24]), so we do not only provide the fastest AND-OR paths but also linear size adders that have a smaller depth by 1.

3.1.4 Organization of this chapter

The remainder of the chapter is organized as follows. In Section 3.2, we describe how instances can be split into parts that can be optimized separately. These sub-

instances include so-called extended And-Or paths that will be introduced in this section. We examine for small values d how many inputs a circuit for an extended And-Or paths with depth d can have. In Section 3.3, we derive a general lower bound on the number of alternating inputs for which we are able to construct a circuit of depth d. Section 3.4 contains the proof of the depth bound of Theorem 3.1.3. In Section 3.5, we present and analyze an algorithm that computes circuits with the properties described in Theorem 3.1.4. In Section 3.6, we modify our algorithm slightly to prove that the circuits we derived can be built with linear size. In Section 3.8, we show practical results of an implementation of the algorithm. Finally, Section 3.9 contains some concluding remarks.

Contributions

The work of this chapter is joint work with Ulrich Brenner. After together discovering the opportunity of an increasing sequence ξ_d, I suggested the first non-explicit sequence ξ_d tending to four, and used it to derive the global depth bound of Theorem 3.1.3 and an asymptotic bound of $\log_2 m + \log_2 \log_2 m - 1$. He suggested the used explicit sequence ξ_d with faster convergence and redid the needed computations that imply the convergence rate to the asymptotic bound provided in Corollary 3.6.30. I did a first analysis to linearize the size of our And-Or paths and to derive linear size adders. We then discussed the details and some improvements together. The implementation of the algorithms, especially of the one used for Section 3.6.4, and most of the writeup is due to myself.

3.2 Recursive splitting of instances

3.2.1 Different splits

The algorithm used to prove our theorems is based on recursive splitting of And-Or path instances into smaller parts that can be realized separately in a recursive manner. The sub-instances that arise are not necessarily And-Or paths but can be described as extended And-Or paths. For $s = (s_0, \ldots, s_{n-1})$, let $\textsc{And}(s) = s_0 \wedge \cdots \wedge s_{n-1}$ and $\textsc{Or}(s) = s_0 \vee \cdots \vee s_{n-1}$.

Definition 3.2.1. Let $m, n \in \mathbb{Z}_{\geq 0}$ and $s = (s_0, \ldots, s_{n-1})$ and $t = (t_0, \ldots, t_{m-1})$ be Boolean variables. An **extended And-Or path** on m inputs is recursively defined as

$$f(s,t) \coloneqq \begin{cases} \textsc{And}(s) & m = 0 \\ \textsc{And}(s) \wedge g(t) & m \geq 1 \end{cases}.$$

As before, its dual is defined by

$$f^*(s,t) = \begin{cases} \textsc{Or}(s) & m = 0 \\ \textsc{Or}(s) \vee g^*(t) & m \geq 1 \end{cases}.$$

Both for $f(s,t)$ and for $f^*(s,t)$, we call $s_0, \ldots, s_{n-1}$ the **symmetric inputs** and $t_0, \ldots, t_{m-1}$ the **alternating inputs**.

Note that it suffices to consider the primal And-Or path, and the dual circuit is then implied by duality. Additionally, observe that for the empty extension s (i.e., if $n = 0$), this gives $f(s,t) = g(t)$.

Using this definition we can introduce the splits that we recursively apply to obtain our results. The first splits have been introduced by Khrapchenko [Khr67]

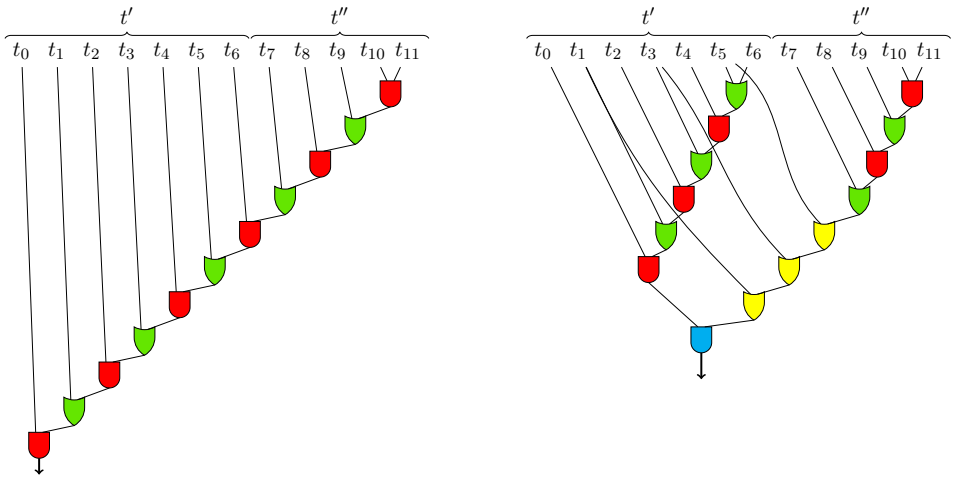
and since then thed have been used by various others, including Grinchuk [Gri09] and Brenner and Silvanus [BS24]. We use the following notation.

Notation 3.2.2. Let $t = (t_0, \ldots, t_{m-1})$ be a tuple of Boolean input variables and let $k \in \mathbb{Z}_{\geq 0}$ be a parameter. We set $t' = (t_0, \ldots, t_{k-1})$ and $t'' = (t_k, \ldots, t_{m-1})$. For any input tuple t we set $\widehat{t} \coloneqq (t_1, t_3, \ldots, t_{m-2})$ if m is odd and $\widehat{t} \coloneqq (t_0, t_2, \ldots, t_{m-2})$ else.

Theorem 3.2.3 (Alternating split with odd prefix)**.** *Let $s = (s_0, \ldots, s_{n-1})$ and $t = (t_0, \ldots, t_{m-1})$ be Boolean input variables and $1 \leq k \leq m$ an odd integer. Using Notation 3.2.2, we have*

$$\begin{aligned} f(s,t) &= f(s,t') \wedge f^*(\widehat{t'}, t'') \qquad \text{and} \\ f^*(s,t) &= f^*(s,t') \vee f(\widehat{t'}, t'') \ . \end{aligned} \tag{3.3}$$

For a proof, see the proof of Lemma 3 from Grinchuk [Gri09] or Lemma 2.5 from Brenner and Hermann [BH19]. An illustration of this split for a standard AND-OR path circuit can be seen in Figures 3.5(a) to 3.5(b).



(a) Standard AND-OR path circuit. **(b)** Circuit after an odd split with $k = 7$.

Figure 3.5: Illustration of an odd split according to Theorem 3.2.3. First, we show the standard circuit for the AND-OR path with $m = 12$. After the split, we implement both arising subfunctions using a standard circuit for the corresponding extended AND-OR paths. The yellow OR gates illustrate the new symmetric prefix and the blue AND gate is the concatenation gate of the two subcircuits.

Notation 3.2.4. For two vectors $(x_0, \ldots, x_{k-1})$ and $(y_0, \ldots, y_{l-1})$ we denote by $(x_0, \ldots, x_{k-1}) \mathbin{+\!\!+} (y_0, \ldots, y_{l-1}) = (x_0, \ldots, x_{k-1}, y_0, \ldots, y_{l-1})$ their concatenation.

Theorem 3.2.5 (Alternating split with even prefix)**.** *Let $s = (s_0, \ldots, s_{n-1})$ and $t = (t_0, \ldots, t_{m-1})$ be Boolean input variables and $1 \leq k \leq m$ an even integer. Then*

$$\begin{aligned} f(s,t) &= f(s,t') \vee f(s \mathbin{+\!\!+} \widehat{t'}, t'') \qquad \text{and} \\ f^*(s,t) &= f^*(s,t') \wedge f^*(s \mathbin{+\!\!+} \widehat{t'}, t'') \ . \end{aligned} \tag{3.4}$$

Proof. For empty $s = ()$, we can use the odd split for the And-Or path shortened by one input and get

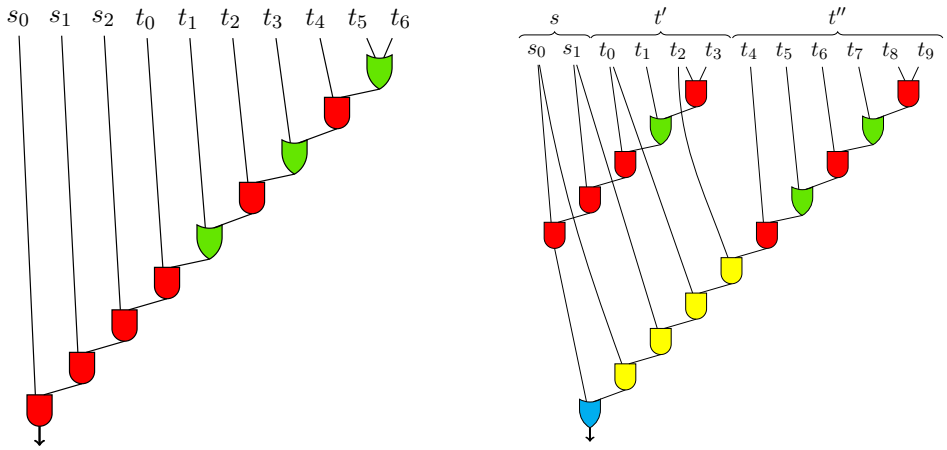
$$\begin{aligned} g(t) &= t_0 \wedge g^*(t_1, \ldots, t_{m-1}) \\ &= t_0 \wedge \Big(g^*(t_1, \ldots, t_{k-1}) \vee f\big((t_2, t_4, \ldots, t_{k-2}), t''\big) \Big) \\ &= g(t') \vee f\big((t_0, t_2, t_4, \ldots, t_{k-2}), t''\big) = g(t') \vee f(\widehat{t'}, t'') \ . \end{aligned}$$

Else, we have

$$(s,t) = \textsc{And}(s) \wedge g(t) = \textsc{And}(s) \wedge \Big(g(t') \vee f(\widehat{t'}, t'') \Big) = f(s,t') \vee f(s +\!\!+ \widehat{t'}, t'') \ .$$

□

An example for an even split can be found in Figure 3.6. In general, the even split is only useful for extended And-Or paths with a comparably short extension. Else, the necessary concatenation of s to the symmetric inputs of the right subcircuit computing $f^*(s +\!\!+ \widehat{t'}, t'')$ makes this subcircuit significantly larger than in the case of the odd split. Nevertheless, there are cases especially for non-extended And-Or paths, in which an even split can be used to achieve a smaller depth.



(a) Standard And-Or path circuit. **(b)** And-Or path circuit after splitting.

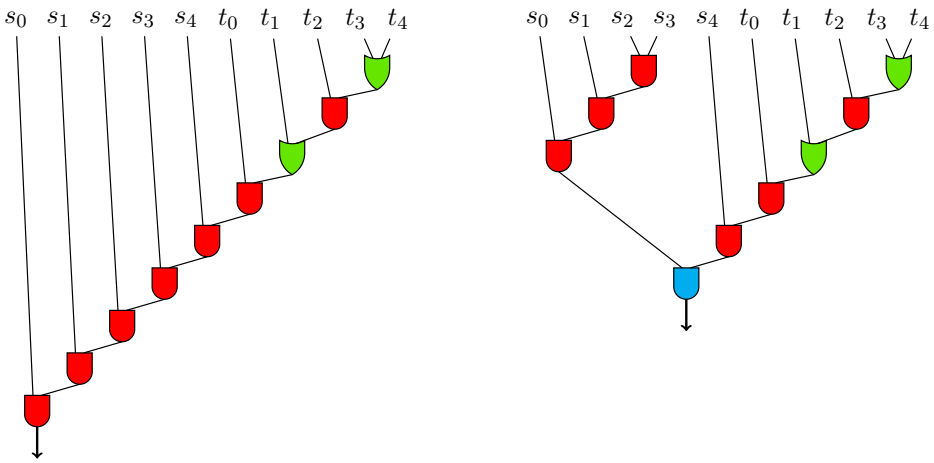
Figure 3.6: Circuits for $f((s_0, s_1), (t_0, \ldots, t_9))$ before and after an even split according to Theorem 3.2.5 for $k = 4$. Once again, the prefix And gates are shown in yellow and the concatenation Or gate is shown in blue

Theorem 3.2.6 (Symmetric split)**.** *Let $s = (s_0, \ldots, s_{n-1})$ and $t = (t_0, \ldots, t_{m-1})$ be Boolean input variables and $1 \leq k \leq n$ an integer. Let $s' = (s_0, \ldots, s_{k-1})$ and $s'' = (s_k, \ldots, s_{n-1})$. Then*

$$\begin{aligned} f(s,t) &= \textit{And}(s') \wedge f(s'', t) \qquad \textit{and} \\ f^*(s,t) &= \textit{Or}(s') \vee f^*(s'', t) \ . \end{aligned} \tag{3.5}$$

An illustration of a symmetric split can be seen in Figure 3.7. This split directly follows from the definition due to the associativity of the And and Or operations.

In contrast to the other splits, there is no additional logic necessary for this split. It just changes the order in which the symmetric inputs and the non-extended AND-OR path are combined. However, building balanced trees for large AND and OR functions nevertheless significantly decreases the depth and is therefore necessary.



(a) Standard AND-OR path circuit. **(b)** AND-OR path circuit after splitting.

Figure 3.7: Circuits for $f((s_0, \dots, s_4), (t_0, \dots, t_4))$ before and after a symmetric split according to Theorem 3.2.6 for $k = 4$.

Lemma 3.2.7. *If $n \leq 2^d$, there is a circuit for $f((s_0 \dots, s_{n-1}), ())$ with depth at most d.*

Proof. Using $f(s, ()) = \text{AND}(s) = \text{AND}(s_0, \dots, s_{\lfloor \frac{n}{2} \rfloor - 1}) \wedge \text{AND}(s_{\lfloor \frac{n}{2} \rfloor}, \dots, s_{n-1})$, such a circuit can be built recursively. Both AND subfunctions can be built with depth $\lceil \log_2(\frac{n}{2}) \rceil \leq d-1$ by induction, and thus, the circuit has depth at most d. Intuitively, such a circuit is just a balanced binary tree with n leaves. □

Note that if $m \leq 2$ then

$$f(s, t) = s_0 \wedge \dots \wedge s_{n-1} \wedge t_0 \wedge \dots \wedge t_{m-1} = f(s +\!\!+ t, ()) \ ,$$

so by the above lemma, if $m + n \leq 2^d$, there is a circuit of depth d for $f((s_0, \dots, s_{n-1}), (t_0, \dots, t_{m-1}))$.

3.2.2 Recursive splitting algorithm

We use these splits to describe an algorithm always preforming the best split. It is summarized in Algorithm 3.1. Algorithms applying recursive splits have already been suggested by Khrapchenko [Khr67], Grinchuk [Gri09], and Hermann [Her20] before. Since nobody yet succeeded in analyzing the best split, neither has used precisely this algorithm but just used the analyzed split. It has been implemented in a generalized version in Brenner and Silvanus [BS23] though. The algorithm works as follows. It is given input strings s and t and its goal is to compute a circuit for the extended AND-OR path $C(s, t)$. Recursively, the algorithm considers all possible splits as introduced in Section 3.2. First, it tries all possible alternating splits, including odd splits as introduced in Theorem 3.2.3 and even splits as in Theorem 3.2.5. Afterward, it tries both reasonable symmetric splits, i.e., splitting off the largest power of two, and

Algorithm 3.1: Optimum splitting of extended And-Or paths

Input: $m, n \in N$ and inputs $s = (s_0, \ldots, s_{n-1})$ and $t = (t_0, \ldots, t_{m-1})$.
Output: a formula circuit for $C(s, t)$.

if $m = 0$ **then**
  return *Optimum circuit for* $\textsc{And}(s_0, \ldots, s_{n-1})$
for $k \in \{1, \ldots, m\}$ **do**
  Let $t' \coloneqq (t_0, \ldots, t_{k-1})$ and $t'' \coloneqq t \setminus t'$.
  $C_1 = \text{OptSplit}(s, t')$
  if k *is odd* **then**
    $C_2 = \text{OptSplit}(\widehat{t'}, t'')$
  else
    $C_2 = \text{OptSplit}(s + \widehat{t'}, t'')$
  if $\text{depth}(C_{n,m}) > \max\{\text{depth}(C_1), \text{depth}(C_2)\} + 1$ **then**
    $C_{n,m} = C_1 \wedge C_2$
Let $s' \subset s$ where $|s'| = 2^{\lfloor \log_2 n \rfloor}$.
$C_1 = \text{OptSplit}(s', ())$
$C_2 = \text{OptSplit}(s \setminus s', t)$
if $\text{depth}(C_{n,m}) > \max\{\text{depth}(C_1), \text{depth}(C_2)\} + 1$ **then**
  $C_{n,m} = C_1 \wedge C_2$
$C_1 = \text{OptSplit}(s, ())$
$C_2 = \text{OptSplit}((), t)$
if $\text{depth}(C_{n,m}) > \max\{\text{depth}(C_1), \text{depth}(C_2)\} + 1$ **then**
  $C_{n,m} = C_1 \wedge C_2$
return $C_{n,m}$

splitting off the whole prefix. For each split it recursively applies the algorithm to find the minimum split circuit for both computed subfunctions. After having considered all the splits, it returns a split that achieves the minimum depth. Table 3.1 shows the achieved depths by splitting for $0 \leq m \leq 40$ and $0 \leq n \leq 60$.

Theorem 3.2.8. *Algorithm 3.1 computes a circuit attaining the minimum depth achievable by splits and it can be implemented in* $\mathcal{O}(m(m+n)\log m)$.

Proof. The algorithm tries all possible splits and thus indeed achieves the minimum depth that can be achieved by splits. To obtain a small running time, we need to bound the number of recursive calls to the algorithm. For this, we ensure that we save all recursive results for the algorithm for every pair (n', m') for which we apply it. Thereby, we only need to call the algorithm once per such pair. Note that whenever we split a circuit into two parts, the number of inputs of both subcircuits is bounded by $m + n$. Thus, we do not need to determine the depth of circuits with more than $m + n$ inputs. Moreover, the number of alternating inputs will never increase, and therefore, this will be at most m. Thus, there are at most $m(m + n)$ recursive calls to the algorithm.

In every recursive call, one can find the minimum depth for inputs (n', m') in logarithmic time: both symmetric splits can be checked in constant time. The optimum k for both an even and for an odd alternating split can be found as follows.

m \ n	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60
0	0	0	1	2	2	3	3	3	3	4	4	4	4	4	4	4	4	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
1	0	1	2	2	3	3	3	3	4	4	4	4	4	4	4	4	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
2	1	2	2	3	3	3	3	4	4	4	4	4	4	4	4	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
3	2	2	3	3	3	3	4	4	4	4	4	4	4	4	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
4	3	3	3	3	4	4	4	4	4	4	4	4	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	7
5	3	3	4	4	4	4	4	4	4	4	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	7	7	7
6	3	3	4	4	4	4	4	4	4	4	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	7	7	7	7
7	4	4	4	4	4	4	4	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	7	7	7	7	7	7	7
8	4	4	4	4	4	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	7	7	7	7	7	7	7	7
9	4	4	4	4	4	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	7	7	7	7	7	7	7	7
10	4	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	7	7	7	7	7	7	7	7	7	7
11	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	7	7	7	7	7	7	7	7	7	7	7	7
12	5	5	5	5	5	5	5	5	5	5	5	5	5	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	7	7	7	7	7	7	7	7	7	7	7	7	7
13	5	5	5	5	5	5	5	5	5	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	7	7	7	7	7	7	7	7	7	7	7	7	7	7
14	5	5	5	5	5	5	5	5	5	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7
15	5	5	5	5	5	5	5	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7
16	5	5	5	5	5	5	5	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7
17	5	5	5	5	5	5	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7
18	5	5	5	5	5	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7
19	5	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7
20	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7
21	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7
22	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7
23	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7
24	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7
25	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7
26	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7
27	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7
28	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7
29	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7
30	6	6	6	6	6	6	6	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7
31	6	6	6	6	6	6	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7
32	6	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7
33	6	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7
34	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7
35	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	8	8	8	8	8	8
36	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	8	8	8	8	8
37	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	8	8	8	8	8
38	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	8	8	8	8	8	8	8
39	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	8	8	8	8	8	8
40	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	8	8	8	8	8	8

Table 3.1: This table shows the optimum depths that can be achieved by splitting along the best split. Every color corresponds to a different depth.

Whenever you consider some k, you can check which of the two subcircuits C_1 or C_2 gives the larger depth. If C_1 has the larger depth, the optimum k cannot be larger than the current k because the depth of C_1 is monotonically increasing with k. If C_2 has the larger depth, the optimum k cannot be smaller than the current because the depth of C_2 is monotonically decreasing with k. Thus, you can for every k check whether the optimum k is larger or smaller and use this to do binary search on the possible values of k. Hence, it takes a logarithmic number of steps in m. This implies a running time of $\mathcal{O}(\log_2 m)$ per recursion step, giving a total running time of $\mathcal{O}(m(m+n)\log m))$. □

To apply this algorithm in practice there are different bounds to avoid considering tuples (n', m') that are not needed. In addition, one can often find the best alternating split in constant time by trying the best achieved splits for one symmetric input less, and one alternating input less. Using such techniques, this algorithm can be applied for very large instances.

3.2.3 Results of optimum splitting

depth	suitable m	$d - \log_2 m - \log_2 \log_2 m$
2	$3 \leq m \leq 3$	-0.24
3	$4 \leq m \leq 6$	0
4	$7 \leq m \leq 10$	-0.29
5	$11 \leq m \leq 19$	-0.24
6	$20 \leq m \leq 33$	-0.43
7	$34 \leq m \leq 60$	-0.43
8	$61 \leq m \leq 109$	-0.49
9	$110 \leq m \leq 202$	-0.54
10	$203 \leq m \leq 376$	-0.6
11	$377 \leq m \leq 699$	-0.65
12	$700 \leq m \leq 1313$	-0.69
13	$1314 \leq m \leq 2466$	-0.73
14	$2467 \leq m \leq 4646$	-0.76
15	$4647 \leq m \leq 8783$	-0.78
16	$8784 \leq m \leq 16629$	-0.81
17	$16630 \leq m \leq 31548$	-0.83
18	$31549 \leq m \leq 60059$	-0.84
19	$60060 \leq m \leq 114650$	-0.86
20	$114651 \leq m \leq 219166$	-0.87

Table 3.2: Depth obtained by iteratively splitting. For each depth d we show the length m of And-Or paths that lead to this depth by iterative splitting. In addition, we show the difference $d - \log_2 m - \log_2 \log_2 m$ for the smallest m within the interval, i.e., leading to the largest constant.

Table 3.2 shows the optimum depths for up to 500,000 inputs that can be achieved by splitting. This algorithm is conjectured to not only yield the best circuit that can be achieved by splitting but to yield an optimum circuit (see Hermann [Her20]). On all instances for which the optimum solution is known due to the optimum algorithm

of Brenner, Silvanus, and Silvanus [BSS22], this algorithm achieves the optimum depth.

In order to analyze the depth of this algorithm, both Grinchuk [Gri09] and Brenner and Silvanus [BS24] analyzed a precisely described split. This split of course depends on m, n and the depth to achieve. It is not necessarily the optimum one, but instead it can be described by precisely giving the value for k. We will take a similar approach. In order to achieve a good result, we nevertheless use this optimum algorithm for small cases. Small circuits are the most important ones because they arise multiple times as subcircuits in our instance. Losing some additional depth in small instances might easily increase the depth of the total circuit.

Table 3.3 shows the maximum number of symmetric inputs that can be part of a circuit for m alternating inputs with depth d computed with Algorithm 3.1. In every cell, we show the number of symmetric inputs as the first number and as the second entry, we show an abbreviation for the split that can be applied to achieve this depth.

For example, if one wants to build a circuit of depth 6 with 25 alternating inputs, Table 3.3 indicates that there may be up to 23 symmetric inputs part of this circuit. The circuit itself is then obtained by an alternating split for $k = 7$. Such a split needs a circuit of depth 5 for $(n, m) = (23, 7)$ and for $(n, m) = (3, 18)$. Indeed, the first circuit can be obtained by a symmetric split with prefix length 16 whereas the second circuit is obtained by an alternating split for $k = 9$. Note that the second subcircuit could even have taken two additional symmetric inputs which is why the entry for depth 5 at $m = 18$ is 5 and not 3. Both of these subcircuit splits can be checked recursively. Eventually, every subcircuit is reduced to a couple of instances of depth 1 which can clearly be realized if they have at most two inputs.

Observation 3.2.9. Table 3.3 shows for all (m, d) where $0 \leq m \leq 33$ and $0 \leq d \leq 6$ a number of symmetric inputs n such that a circuit of depth d for $f((s_0, \ldots, s_{n-1}), (t_0, \ldots, t_{m-1}))$ exists.

3.3 Bounding the number of alternating inputs in circuits

We start to bound the depth of our circuits by first introducing a bound on the number of alternating inputs that can be part of a circuit for a given depth and a given number of symmetric inputs. We define a bound $\mu(d, n)$ and we will prove that whenever $m \leq \mu(d, n)$ for $d \geq 6$, then there exists a circuit for (n, m) with depth d. This inverse approach is suitable because such a number can be used to determine a suitable k for an alternating split achieving depth d. Later, we use this bound $\mu(d, n)$ to instead bound d in terms of n and m.

Definition 3.3.1. Let the sequence $(\xi_d)_{d \in \mathbb{Z}_{\geq 1}}$ be given by $\xi_d = \left(4 - \frac{3}{\sqrt{d}}\right)$ for all $d \in \mathbb{Z}_{\geq 1}$. For $d \in \mathbb{Z}_{\geq 1}$ and $n \in \mathbb{Z}_{\geq 0}$, we set

$$\mu(d, n) := \begin{cases} 0 & \text{if } n \geq 2^d \ , \\ 1 & \text{if } 2^d > n \geq 2^d - 2 \ , \\ \xi_d \frac{2^d - n - 2}{d} + 3 & \text{otherwise } . \end{cases}$$

Note that a similar approach has already been used by Grinchuk [Gri09] as well as the other approaches mentioned in Figure 3.4 before. Grinchuk introduced a bound such that not only $\lfloor \mu \rfloor$ forms a lower bound, but already μ does. Their proof was simplified and the bound was changed to $\mu(d, n) = \xi \frac{2^d - n - 2}{d} + 2$. Hermann [Her20]

m \ d	1	2		3		4		5		6	
0	2	4	S2	8	S4	16	S8	32	S16	64	S32
1	1	3	S2	7	S4	15	S8	31	S16	63	S32
2	0	2	S2	6	S4	14	S8	30	S16	62	S32
3		1	A1	5	S4	13	S8	29	S16	61	S32
4				3	A1	11	S8	27	S16	59	S32
5				1	A3	9	S8	25	S16	57	S32
6				1	A3	9	S8	25	S16	57	S32
7						7	A1	23	S16	55	S32
8						5	A3	21	S16	53	S32
9						5	A3	21	S16	53	S32
10						0	A6	16	S16	48	S32
11								15	A1	47	S32
12								13	A3	45	S32
13								9	A5	41	S32
14								9	A5	41	S32
15								7	A7	39	S32
16								7	A7	39	S32
17								5	A9	37	S32
18								5	A9	37	S32
19								0	A10	32	S32
20										31	A1
21										29	A3
22										25	A5
23										25	A5
24										23	A7
25										23	A7
26										21	A9
27										21	A9
28										15	A11
29										15	A11
30										7	A15
31										7	A15
32										0	A18
33										0	A19

Table 3.3: This table shows for each number m of alternating inputs and each depth d two entries. The first one is the maximum number n of symmetric inputs that an extended And-Or path circuit with m alternating inputs and depth d that is computed by our algorithm can contain. The second entry is the split leading to this depth on the right. Split *Ak* stands for an alternating split with parameter k whereas *Sk* stands for a symmetric split with prefix length k.

used this function for $\xi = 1$ and later Brenner and Silvanus [BS24] improved this to $\xi = 1.999$. This ξ is tight in the sense that it cannot be increased to 2 or beyond 2 without losing the property of being such a lower bound. To improve on their result, we use two new ideas. First, we allow for our bound to not hold for small $d \leq 5$. Thereby, we can increase our bound, and we check small cases by hand. Second, we do not use a constant ξ but instead a sequence that increases with the depth. This allows us to analyze the asymptotic behavior of the depth in our algorithm better than with a constant bound.

We start by showing some properties of ξ_d and $\mu(d, n)$ that we need later.

Observation 3.3.2. The sequence $(\xi_d)_{d \in \mathbb{Z}_{\geq 1}}$ is monotonically increasing and its limit is $\lim_{d \to \infty} \xi_d = 4$.

Lemma 3.3.3. *The sequence $(\xi_d)_{d \in \mathbb{Z}_{\geq 1}}$ satisfies the following bounds on its increase rate.*

(i) $\frac{\xi_{d+1}}{\xi_d} - 1 \leq \frac{3}{4-\frac{3}{\sqrt{d}}} \frac{1}{2d\sqrt{d+1}}$ *for all* $d \in \mathbb{Z}_{\geq 1}$.

(ii) $\xi_{d+1} \leq \frac{d+1}{d}\xi_d$ *for all* $d \in \mathbb{Z}_{\geq 2}$.

Proof.

Claim. For $d \in \mathbb{Z}_{\geq 1}$ we have

$$\frac{1}{\sqrt{d}} - \frac{1}{\sqrt{d+1}} < \frac{1}{2d\sqrt{d+1}} \ . \tag{3.6}$$

Proof of claim. To see this, note that the inequality is equivalent to

$$\sqrt{\frac{d+1}{d}} < \frac{2d+1}{2d}$$

and

$$\frac{4d^2(d+1)}{d} < 4d^2 + 4d + 1 \ .$$

This is equivalent to

$$4d^3 + 4d^2 < 4d^3 + 4d^2 + d \ ,$$

which is obviously true. This shows the claim. □

(i) We have

$$\begin{aligned}
\left(\frac{\xi_{d+1}}{\xi_d} - 1\right) &= \left(\frac{4 - \frac{3}{\sqrt{d+1}}}{4 - \frac{3}{\sqrt{d}}} - 1\right) \\
&= \frac{1}{4 - \frac{3}{\sqrt{d}}}\left(4 - \frac{3}{\sqrt{d+1}} - 4 + \frac{3}{\sqrt{d}}\right) \\
&= \frac{3}{4 - \frac{3}{\sqrt{d}}}\left(\frac{1}{\sqrt{d}} - \frac{1}{\sqrt{d+1}}\right) \\
&\overset{(3.6)}{<} \frac{3}{4 - \frac{3}{\sqrt{d}}} \frac{1}{2d\sqrt{d+1}} \ .
\end{aligned}$$

(ii) We compute

$$\begin{aligned}
\frac{d+1}{d}\xi_d - \xi_{d+1} &= \frac{d+1}{d}\left(4 - \frac{3}{\sqrt{d}}\right) - 4 + \frac{3}{\sqrt{d+1}} \\
&= 4 - \frac{3}{\sqrt{d}} + \frac{4}{d} - \frac{3}{d\sqrt{d}} - 4 + \frac{3}{\sqrt{d+1}} \\
&= 3\left(\frac{1}{\sqrt{d+1}} - \frac{1}{\sqrt{d}}\right) + \frac{4}{d} - \frac{3}{d\sqrt{d}} \\
&\overset{(3.6)}{>} -\frac{3}{2d\sqrt{d+1}} + \frac{4}{d} - \frac{3}{d\sqrt{d}} \\
&\geq -\frac{3}{2d\sqrt{d}} + \frac{4}{d} - \frac{3}{d\sqrt{d}} \\
&= \frac{1}{d}\left(4 - \frac{4.5}{\sqrt{d}}\right) \\
&\overset{d\geq 2}{\geq} 0 \ ,
\end{aligned}$$

which shows the statement. □

Lemma 3.3.4. *Let $d \geq 6$ and $n \in \mathbb{Z}_{\geq 0}$. Then $\mu(d,n) < 2^d$.*

Proof. If $n \geq 2^d - 2$, we have $\mu(d,n) \leq 1 < 2^d$, so assume $n < 2^d - 2$, which implies $\mu(d,n) = \xi_d \frac{2^d - n - 2}{d} + 3$. For $d \geq 6$, we have $d > 1.5\xi_d$, so we get

$$\mu(d,n) = \xi_d \frac{2^d - n - 2}{d} + 3 \leq \xi_d \frac{2^d - 2}{d} + 3 < \frac{2^{d+1} - 4}{3} + 3 < 2^d \ .$$

To see the last inequality, note that it is equivalent to $2^{d+1} < 1.5 \cdot 2^{d+1} - 5$. □

Observation 3.3.5. Let $d, n \in \mathbb{Z}_{\geq 0}$ such that $d \geq 1$ and $2 \leq \mu(d,n)$. Then $n < 2^d - 2$.

These have been the properties we need later for $\mu(d,n)$ and $(\xi_d)_{d \in \mathbb{Z}_{\geq 1}}$. Next, we note that even though the defined function $\mu(d,n)$ does not imply a circuit of depth d for small d in general, we can at least find some circuit of depth at most 6 when $\mu(d,n)$ indicates this. This is important because we want to prove the statement that $\mu(d,n)$ implies a circuit by induction and this is needed for the start.

Lemma 3.3.6. *Let $n \in \mathbb{Z}_{\geq 0}$ such that $0 \leq n < 2^6$ and $m \leq \mu(6,n)$. Then there exists a circuit for the extended* And-Or *path $f((s_0, \ldots, s_{n-1}), (t_0, \ldots, t_{m-1}))$ with depth at most 6.*

Proof. This is just a case distinction, and we use Table 3.3 to prove it. For every possible $0 \leq n \leq 63$, we give the value of $\mu(6,n)$ in Table 3.4. Moreover, we give the maximum m that is implied by Table 3.3. More precisely, given d and n we find the maximum m for which n symmetric inputs can be added to the circuit for depth d. This m is listed in Table 3.4 as "max. m". As we always have $\lfloor \mu(6,n) \rfloor \leq m$, we can always apply the splits given by Table 3.3 to achieve the depths required by the μ-values. □

We continue by proving that a circuit exists for small values of m.

Lemma 3.3.7. *Let $d, n, m \in \mathbb{Z}_{\geq 0}$ such that $d \geq 1$, $0 \leq n < 2^d$ and $m \leq \mu(d,n)$. If $m \leq 2$, 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. $f((s_0, \ldots, s_{n-1}), (t_0, \ldots, t_{m-1}))$ is symmetric for $m \leq 2$. Thus, it suffices to prove that there are in total at most 2^d inputs and then use Lemma 3.2.7. Since $m \leq 2$ and $n < 2^d$, we only have to rule out the case that $n = 2^d - 1$. Else, $m + n \leq 2^d$ anyway. But if $n = 2^d - 1$, then, we have $\mu(d,n) = 1$, which shows $m \leq 1$ and, again, $m + n \leq 2^d$. □

As main ingredient for the induction we need the following lemma to ensure that we can apply an alternating split if necessary. It ensures that in case of an alternating split, our bound given by μ implies that both subcircuits have the required depth.

Lemma 3.3.8. *Let $d, n, m, k \in \mathbb{Z}_{\geq 0}$ such that $d \geq 6$, $m \leq \mu(d+1, n)$ and $\mu(d,n) \geq 1$, and let k be the maximum odd integer such that $k \leq \mu(d,n)$. Then $0 \leq \frac{k-1}{2} < 2^d$ and*

$$m - k \leq \mu\left(d, \frac{k-1}{2}\right) \ . \tag{3.7}$$

n	max. m	$\mu(6,n)$
0	33	31.68
1	31	31.22
2	31	30.76
3	31	30.3
4	31	29.83
5	31	29.37
6	31	28.91
7	31	28.44
8	29	27.98
9	29	27.52
10	29	27.06
11	29	26.59
12	29	26.13
13	29	25.67
14	29	25.21
15	29	24.74
16	27	24.28
17	27	23.82
18	27	23.36
19	27	22.89
20	27	22.43
21	27	21.97
22	25	21.51
23	25	21.04
24	23	20.58
25	23	20.12
26	21	19.66
27	21	19.19
28	21	18.73
29	21	18.27
30	20	17.81
31	20	17.34
32	19	16.88
33	18	16.42
34	18	15.96
35	18	15.49
36	18	15.03
37	18	14.57
38	16	14.11
39	16	13.64
40	14	13.18
41	14	12.72
42	12	12.26
43	12	11.79
44	12	11.33
45	12	10.87
46	11	10.41
47	11	9.94
48	10	9.48
49	9	9.02
50	9	8.56
51	9	8.09
52	9	7.63
53	9	7.17
54	7	6.71
55	7	6.24
56	6	5.78
57	6	5.32
58	4	4.86
59	4	4.39
60	3	3.93
61	3	3.47
62	2	1
63	1	1

Table 3.4: Maximum number of alternating inputs in the circuits computed by our algorithm and values of $\mu(6,n)$ (for all values of n, for which $\mu(6,n) > 0$). One can check that $\lfloor \mu(6,n) \rfloor$ is indeed a lower bound for the maximum number m of inputs by checking that the difference between the middle and the right column is always larger than -1.

Proof. We have $\frac{k-1}{2} \leq \frac{\mu(d,n)-1}{2} \leq \frac{2^d-1}{2} < 2^d - 2 < 2^d$ using Lemma 3.3.4.

So it remains to show Equation (3.7). As we have $m \leq \mu(d+1,n)$, it suffices to show that

$$\mu\left(d, \frac{k-1}{2}\right) + k \geq \mu(d+1,n) \ .$$

To prove this, we need to know in which case of Definition 3.3.1 we are for each the argument sets of μ. As noted above, we have $\frac{k-1}{2} < 2^d - 2$. Therefore, it is

$$\mu\left(d, \frac{k-1}{2}\right) = \xi_d \frac{2^d - \frac{k-1}{2} - 2}{d} + 3 \ . \tag{3.8}$$

Moreover, we have $\mu(d,n) \geq 1$, so $n < 2^d < 2^{d+1} - 2$. Thus, we get

$$\mu(d+1,n) = \xi_{d+1} \frac{2^{d+1} - n - 2}{d+1} + 3 \ . \tag{3.9}$$

We use both of the equations to bound the difference of the μ values. This implies

$$\begin{aligned}
&\mu\left(d, \frac{k-1}{2}\right) + k - \mu(d+1, n) \qquad (3.10)\\
\overset{(3.8)}{=}\ & \xi_d \frac{2^d - \frac{k-1}{2} - 2}{d} + 3 + k - \mu(d+1, n)\\
=\ & k\left(1 - \frac{\xi_d}{2d}\right) + \xi_d \frac{2^d - 2}{d} + \frac{\xi_d}{2d} + 3 - \mu(d+1, n)\\
\overset{(3.9)}{=}\ & k\left(1 - \frac{\xi_d}{2d}\right) + \xi_d \frac{2^d - 2}{d} + \frac{\xi_d}{2d} - \xi_{d+1}\frac{2^{d+1} - n - 2}{d+1}\\
>\ & (\mu(d, n) - 2)\left(1 - \frac{\xi_d}{2d}\right) + \xi_d \frac{2^d - 2}{d} + \frac{\xi_d}{2d} - \xi_{d+1}\frac{2^{d+1} - n - 2}{d+1}\ .
\end{aligned}$$

Here, we need to do a case distinction whether $n < 2^d - 2$ or not to know the value of $\mu(d, n)$.

Case 1. $n < 2^d - 2$

Proof. Then, we have $\mu(d, n) = \xi_d \frac{2^d - n - 2}{d} + 3$. Moreover,

$$\xi_d \geq 4 - \frac{3}{\sqrt{6}} \geq 4 - \frac{3}{2} > 2\ .$$

Thus, we have

$$\begin{aligned}
\frac{\frac{\xi_{d+1}}{\xi_d}}{d+1} - \frac{1}{d} + \frac{\xi_d}{2d^2} \quad &= \quad \frac{2d^2 \frac{\xi_{d+1}}{\xi_d} - 2d(d+1) + \xi_d(d+1)}{2d^2(d+1)}\\
&\overset{\xi_{d+1} \geq \xi_d}{\geq} \quad \frac{-2d + \xi_d(d+1)}{2d^2(d+1)}\\
&\overset{\xi_d \geq 2}{\geq} \quad 0\ . \qquad (3.11)
\end{aligned}$$

Thus, Equation (3.10) leads to

$$\begin{aligned}
&\mu\left(d, \frac{k-1}{2}\right) + k - \mu(d+1, n)\\
>\ & \left(\xi_d \frac{2^d - n - 2}{d} + 1\right)\left(1 - \frac{\xi_d}{2d}\right) + \xi_d \frac{2^d - 2}{d} + \frac{\xi_d}{2d} - \xi_{d+1}\frac{2^{d+1} - n - 2}{d+1}\\
=\ & \xi_d \frac{2^d - 2}{d}\left(2 - \frac{\xi_d}{2d}\right) + \left(1 - \frac{\xi_d}{2d}\right) + \frac{\xi_d}{2d} - \xi_{d+1}\frac{2^{d+1} - 2}{d+1}\\
& + n\xi_d\left(\frac{\frac{\xi_{d+1}}{\xi_d}}{d+1} - \frac{1}{d} + \frac{\xi_d}{2d^2}\right)\\
\overset{(3.11)}{\geq}\ & \xi_d \frac{2^d - 2}{d}\left(2 - \frac{\xi_d}{2d}\right) + 1 - \xi_{d+1}\frac{2^{d+1} - 2}{d+1}\\
=\ & \frac{\xi_d}{2d^2(d+1)} \underbrace{\left((2^d - 2)(4d - \xi_d)(d+1) + \frac{2d^2(d+1)}{\xi_d} - 4\frac{\xi_{d+1}}{\xi_d}(2^d - 1)d^2\right)}_{=:\alpha(d)}.
\end{aligned}$$

We show the nonnegativity of $\alpha(d)$.

$$\begin{aligned}\alpha(d) &\overset{\xi_{d+1}\geq\xi_d}{\geq} 2^d\left(4d^2+(4-\xi_d)d-\xi_d-\frac{\xi_{d+1}}{\xi_d}4d^2\right)-8d^2-8d+2\xi_d d+2\xi_d \\ &\quad + \frac{2d^3+2d^2}{\xi_d}+4d^2 \\ &= 2^d\left(4d^2+(4-\xi_d)d-\xi_d-\frac{\xi_{d+1}}{\xi_d}4d^2\right)-4d^2-8d+2\xi_d d+2\xi_d \\ &\quad + \frac{2d^3+2d^2}{\xi_d}\end{aligned} \tag{3.12}$$

We now use inequality (i) from Lemma 3.3.3.

$$\begin{aligned}4d^2+(4-\xi_d)d-\xi_d-\frac{\xi_{d+1}}{\xi_d}4d^2 &= -4\left(\frac{\xi_{d+1}}{\xi_d}-1\right)d^2+\frac{3}{\sqrt{d}}d-4+\frac{3}{\sqrt{d}} \\ &\overset{(i)}{\geq} -4\frac{3}{4-\frac{3}{\sqrt{d}}}\frac{1}{2d\sqrt{d+1}}d^2+\frac{3}{\sqrt{d}}d-4+\frac{3}{\sqrt{d}} \\ &= -\frac{6}{4-\frac{3}{\sqrt{d}}}\frac{1}{\sqrt{d+1}}d+\frac{3}{\sqrt{d}}d-4+\frac{3}{\sqrt{d}} \\ &= 3\sqrt{d}\left(1-\frac{2}{4-\frac{3}{\sqrt{d}}}\sqrt{\frac{d}{d+1}}\right)-4+\frac{3}{\sqrt{d}}\end{aligned}$$

Thus, it is sufficient to show that

$$2^d\left(3\sqrt{d}\left(1-\frac{2}{4-\frac{3}{\sqrt{d}}}\sqrt{\frac{d}{d+1}}\right)-4+\frac{3}{\sqrt{d}}\right)-4d^2-8d+2\xi_d d+2\xi_d+\frac{2d^3+2d^2}{\xi_d}\geq 0\ . \tag{3.13}$$

For $d\in\{6,7\}$, this is easily checked.

For $d=6$ we use that $\sqrt{6}\in(2.44,2.45)$ and thus $\xi_6\in(2.77,2.8)$. Moreover, we have $\sqrt{\frac{6}{7}}\leq 0.93$, so for $d=6$ the left side of Equation (3.13) is upper bounded by

$$\begin{aligned}&\geq 64(7.3(1-2/2.77\cdot 0.93)-4+1.2)-144-48+14\cdot 2.7+\frac{432+72}{2.8} \\ &\geq 64(2.39-4+1.22)-192+37+180 \\ &\geq -25+25 \\ &= 0\ .\end{aligned}$$

Similarly, for $d=7$ we have that $\sqrt{7}\in(2.64,2.65)$ and thus $\xi_7\in(2.86,2.9)$. Moreover, we have $\sqrt{\frac{7}{8}}\leq 0.94$, so for $d=7$ the left side of Equation (3.13) is upper bounded by

$$\begin{aligned}&\geq 128(7.9(1-2/2.86\cdot 0.94)-4+1.13)-196-56+16\cdot 2.8+\frac{686+98}{2.9} \\ &\geq 128(2.7-4+1.13)-252+44+270 \\ &\geq -22+62 \\ &> 0\ .\end{aligned}$$

This concludes the calculation for $d \in \{6, 7\}$, so assume $d \geq 8$.

Claim. For $d \geq 8$, we have

$$2^d \left(3\sqrt{d} \left(1 - \frac{2}{4 - \frac{3}{\sqrt{d}}} \sqrt{\frac{d}{d+1}} \right) - 4 + \frac{3}{\sqrt{d}} \right) \geq 0 \ . \tag{3.14}$$

Proof of claim. To prove the claim, consider:

$$\begin{aligned} 3\sqrt{d} \left(1 - \frac{2}{4 - \frac{3}{\sqrt{d}}} \sqrt{\frac{d}{d+1}} \right) - 4 + \frac{3}{\sqrt{d}} &\geq 3\sqrt{d} \left(1 - \frac{2}{4 - \frac{3}{\sqrt{8}}} \sqrt{\frac{d}{d+1}} \right) - 4 + \frac{3}{\sqrt{d}} \\ &\geq 3\sqrt{d} \left(1 - 0.69 \sqrt{\frac{d}{d+1}} \right) - 4 + \frac{3}{\sqrt{d}} \ . \end{aligned}$$

If $d \geq 12$, then

$$3\sqrt{d} \left(1 - 0.69 \sqrt{\frac{d}{d+1}} \right) - 4 + \frac{3}{\sqrt{d}} \geq 3\sqrt{d}(1 - 0.69) - 4 + \frac{3}{\sqrt{d}} = 0.93\sqrt{d} - 4 + \frac{3}{\sqrt{d}} \ .$$

This term is non-negative because the derivative of $x \mapsto 0.93\sqrt{x} - 4 + \frac{3}{\sqrt{x}}$ is $\frac{0.93x - 3}{2x^{1.5}}$, which is positive for $x \geq 3.3$, and we have $0.93\sqrt{12} - 4 + \frac{3}{\sqrt{12}} \geq 0.08 > 0$.

If $9 \leq d \leq 11$, then

$$3\sqrt{d} \left(1 - 0.69 \sqrt{\frac{d}{d+1}} \right) - 4 + \frac{3}{\sqrt{d}} \geq 3\sqrt{d} \left(1 - 0.69 \sqrt{\frac{11}{12}} \right) - 4 + \frac{3}{\sqrt{d}} = \sqrt{d} - 4 + \frac{3}{\sqrt{d}} \ .$$

This term is non-negative because the derivative of $x \mapsto \sqrt{x} - 4 + \frac{3}{\sqrt{x}}$ is $\frac{x-3}{2x^{1.5}}$, which is positive for $x > 3$, and we have $\sqrt{9} - 4 + \frac{3}{\sqrt{9}} = 0$.

Finally, if $d = 8$, we have

$$3\sqrt{8} \left(1 - 0.69 \sqrt{\frac{8}{9}} \right) - 4 + \frac{3}{\sqrt{8}} \geq 0.02 > 0 \ .$$

This proves the claim, i.e., Equation (3.14). □

For the other part of Equation (3.13) we have (since $\xi_d \in [2.9, 4]$ for $d \geq 8$):

$$\begin{aligned} & - 4d^2 - 8d + 2\xi_d d + 2\xi_d + \frac{2d^3 + 2d^2}{\xi_d} \\ = \ & 2\xi_d + d \left(d \left(\frac{2d}{\xi_d} + \frac{2}{\xi_d} - 4 \right) + 2\xi_d - 8 \right) \\ \geq \ & 2 \cdot 2.9 + d \left(d \left(\frac{2 \cdot 8}{4} + \frac{2}{4} - 4 \right) + 2 \cdot 2.9 - 8 \right) \\ = \ & 5.8 + d(0.5d - 2.2) \\ \geq \ & 0 \ . \end{aligned}$$

Together with Equation (3.14), this implies Equation (3.13). □

Case 2. $n \geq 2^d - 2$.

Proof. In this case, we have $\mu(d,n) = 1$. Using Equation (3.10), we get

$$\begin{aligned}
& \mu\left(d, \frac{k-1}{2}\right) + k - \mu(d+1,n) \\
> \; & -\left(1 - \frac{\xi_d}{2d}\right) + \xi_d \frac{2^d - 2}{d} + \frac{\xi_d}{2d} - \xi_{d+1} \frac{2^{d+1} - n - 2}{d+1} \\
\geq \; & -\left(1 - \frac{\xi_d}{2d}\right) + \xi_d \frac{2^d - 2}{d} + \frac{\xi_d}{2d} - \xi_{d+1} \frac{2^{d+1} - 2^d + 2 - 2}{d+1} \\
= \; & \xi_d \frac{2^d - 2}{d} - 1 + \frac{\xi_d}{d} - \xi_{d+1} \frac{2^d}{d+1} \\
= \; & \frac{\xi_d}{d(d+1)} \underbrace{\left(\left(2^d - 2\right)(d+1) - \frac{d(d+1)}{\xi_d} + (d+1) - \frac{\xi_{d+1}}{\xi_d} 2^d d\right)}_{=:\alpha(d)}.
\end{aligned}$$

We show the nonnegativity of $\alpha(d)$.

$$\begin{aligned}
\alpha(d) \quad &= \quad 2^d \left(d + 1 - \frac{\xi_{d+1}}{\xi_d} d\right) - 2d - 2 - \frac{d(d+1)}{\xi_d} + d + 1 \\
&= \quad 2^d \left(1 - \left(\frac{\xi_{d+1}}{\xi_d} - 1\right) d\right) - d - 1 - \frac{d(d+1)}{\xi_d}
\end{aligned}$$

We now use inequality (i) from Lemma 3.3.3:

$$1 - \left(\frac{\xi_{d+1}}{\xi_d} - 1\right) d \overset{(i)}{\geq} 1 - \frac{3}{4 - \frac{3}{\sqrt{d}}} \frac{1}{2d\sqrt{d+1}} d = 1 - \frac{3}{4 - \frac{3}{\sqrt{d}}} \frac{1}{2\sqrt{d+1}} .$$

Thus, using that $2^d \geq 1.5d^2$ for $d \geq 6$, we have

$$\begin{aligned}
\alpha(d) \quad &\geq \quad 2^d \left(1 - \frac{3}{4 - \frac{3}{\sqrt{d}}} \frac{1}{2\sqrt{d+1}}\right) - d - 1 - \frac{d(d+1)}{\xi_d} \\
&\overset{d \geq 6}{\geq} \quad 2^d \left(1 - \frac{3}{4 - \frac{3}{\sqrt{6}}} \frac{1}{2\sqrt{7}}\right) - d - 1 - \frac{d(d+1)}{\xi_d} \\
&\overset{\xi_d \geq 2}{\geq} \quad 0.79 \cdot 2^d - d - 1 - 0.5d^2 - 0.5d \\
&\overset{2^d \geq 1.5d^2}{\geq} \quad 0.79 \cdot 1.5d^2 - 0.5d^2 - 2d \\
&\geq \quad d^2 - 0.5d^2 - 2d \\
&\overset{d \geq 4}{\geq} \quad 0 \; .
\end{aligned}$$

□

This concludes the case distinction and thus proves Lemma 3.3.8. □

Remark 3.3.9. As noted in Observation 3.3.2 the sequence ξ_d tends to four for large d. This is in a sense as good as possible. Looking more closely at the last proof and especially at Equation (3.12), one can see that for $\alpha(d)$ to be positive, the term 2^d needs to have a positive coefficient. This can only be true if $4 - \xi_d \geq 0$ because among the four terms in brackets, the first term is smaller than the last one

and the third summand $-\xi_d$ is negative anyway. This is the reason why ξ_d cannot be chosen at least or even larger than four under an analysis like this. Indeed, assuming that $\xi = 5$, one can construct counter examples to the statement of Lemma 3.3.8 like for example $(n, m, d, k) = (0, 154, 7, 93)$ or $(n, m, d, k) = (0, 900, 10, 513)$, or even larger examples like $(n, m, d, k) = (0, 491524, 20, 262145)$. As this lemma is crucial for the recursion by alternating splits to work, one cannot increase ξ beyond four that easily.

This concludes the ingredients to prove that a circuit exists whenever this is implied by the bound $\mu(d, n)$ for $d \geq 6$.

Proposition 3.3.10. *Let $d, n \in \mathbb{Z}_{\geq 0}$ such that $d \geq 6$ and $0 \leq n < 2^d$. Then for all $m \leq \mu(d, n)$ there exists a circuit for the extended And-Or path $f((s_0, \ldots, s_{n-1}), (t_0, \ldots, t_{m-1}))$ with depth at most d.*

Proof. We do induction on d. The base case $d = 6$ follows from Lemma 3.3.6.

In the induction step, assume that the statement is true for some $d \geq 6$ and for all $0 \leq n < 2^d$. Now given some $n, m \in \mathbb{Z}_{\geq 0}$ with $m \leq \mu(d + 1, n)$ and some inputs $s = (s_0, \ldots, s_{n-1})$ and $t = (t_0, \ldots t_{m-1})$, we need to construct a circuit for $f(s, t)$ of depth at most $d + 1$. If $m \leq 2$, the result directly follows from Lemma 3.3.7. If $3 \leq m \leq \mu(d, n)$ and thus $n \leq 2^d - 2$, then we even obtain a circuit of depth $d < d + 1$ by induction assumption. So for the remaining proof, assume that $m \geq 3$ and $\mu(d, n) < m$.

Case 1. $\mathbf{n \geq 2^d}$

Proof. We apply a symmetric split $f(s, t) = \textsc{And}(s') \wedge f(s \setminus s', t)$, where s' is a subtuple of s of length $k = 2^d$. We need to show that this split is of depth at most $d + 1$. So it suffices to show that both $\textsc{And}(s')$ and $f(s \setminus s', t)$ can be realized with depth d.

A subcircuit for $\textsc{And}(s')$ can be constructed easily as a binary tree with depth d (see Lemma 3.2.7). For the other part, we have

$$|s \setminus s'| = |s| - 2^d < 2^{d+1} - 2 - 2^d = 2^d - 2$$

by Observation 3.3.5.

Since we have both $|s| < 2^{d+1} - 2$ and $|s \setminus s'| < 2^d - 2$, we have

$$\mu(d + 1, n) = \xi_{d+1} \frac{2^{d+1} - n - 2}{d + 1} + 3 \qquad \text{and}$$
$$\mu(d, n - k) = \xi_d \frac{2^d - (n - k) - 2}{d} + 3 \ .$$

We can show that $m \leq \mu(d, n - k)$ as follows by using Lemma 3.3.3 (ii).

$$\begin{aligned}
\mu(d + 1, n) &= \xi_{d+1} \frac{2^{d+1} - n - 2}{d + 1} + 3 \\
&= \xi_{d+1} \frac{2^d - (n - k) - 2}{d + 1} + 3 \\
&= \frac{\xi_{d+1}}{\xi_d} \frac{d}{d + 1} \xi_d \frac{2^d - (n - k) - 2}{d} + 3 \\
&\overset{\text{Lem. 3.3.3 (ii)}}{\leq} \xi_d \frac{2^d - (n - k) - 2}{d} + 3 \\
&= \mu(d, n - k) \ .
\end{aligned}$$

This implies $m \leq \mu(d+1,n) \leq \mu(d,n-k)$ as desired. Hence, by induction hypothesis, we can build $f(s',t)$ with depth d and get a circuit of depth $d+1$ for $f(s,t)$. This proves the proposition in the case $n \geq 2^d$. □

Case 2. $\mathbf{n < 2^d}$

Proof. If $n \geq 2^d - 2$, then $\mu(d,n) = 1$, and if $n < 2^d - 2$ then

$$\mu(d,n) = \xi_d \frac{2^d - n - 2}{d} + 3 \geq \xi_d \frac{1}{d} + 3 > 1 \ .$$

So we get $\mu(d,n) \geq 1$ in all cases, and we can choose a maximum odd integer k such that $k \leq \mu(d,n)$, especially, $k < m$. We apply the odd alternating split $f(s,t) = f(s,t') \wedge f^*(\widehat{t'},t'')$ with prefix length k given by Theorem 3.2.3, where $t' = (t_0,\ldots,t_{k-1})$ and $t'' = (t_k,\ldots,t_{m-1})$. Due to our choice of k, a circuit for $f(s,t')$ with depth d exists by induction. The extended AND-OR path $f^*(\widehat{t'},t'')$ has $|\widehat{t'}| = \frac{k-1}{2}$ symmetric and $|t''| = m-k$ alternating inputs. Due to Lemma 3.3.8 we know in addition that $\frac{k-1}{2} < 2^d$ and $m-k \leq \mu\Big(d, \frac{k-1}{2}\Big)$. Thus, $f(\widehat{t'},t'')$ can be built with depth d by induction, too. In total, this implies a circuit with depth $d+1$ for $f(s,t)$. □
This completes the proof of Proposition 3.3.10. □

Note that the proof we gave is in fact a constructive proof. The circuits that were proven to exist can be constructed by always applying the split for which we proved the corresponding depth bound. Formalizing this as an algorithm yields Algorithm 3.2.

Observation 3.3.11. Let $n,m,d \in \mathbb{Z}_{\geq 0}$ such that $d \geq 6$ and $2 \leq m \leq \mu(d,n)$. Then Algorithm 3.2 computes a circuit of depth at most d when given the depth threshold $d_0 = 6$.

Algorithm 3.2: Depth optimization for extended AND-OR paths

Input: $n,m \in \mathbb{Z}_{\geq 0}$, a set of symmetric inputs $s = (s_0,\ldots,s_{n-1})$, a set of alternating inputs $t = (t_0,\ldots,t_{m-1})$ and a depth threshold d_0.
Output: a circuit $C(s,t)$ computing $f(s,t)$.

```
if m ≤ 2 then
    return Optimum circuit for AND(s_0, ..., s_{n-1}, t_0, ..., t_{m-1})
d ← min{ d | m ≤ μ(d,n) }
if d ≤ d_0 then
    return Optimum circuit for f(s,t)
if n ≥ 2^{d-1} then
    Choose s' ⊆ s such that |s'| = 2^{d-1}.
    Compute an optimum circuit S for AND(s').
    return S ∧ C(s \ s', t)
else
    k ← max{ k | k odd, k ≤ μ(d-1,n) }.
    Let t' ← (t_0, ..., t_{k-1}) and t'' ← t \ t'.
    return C(s,t') ∧ ( C(t̂', t'') )*
```

Note that Algorithm 3.1 also yields a circuit of the same depth because it always applies the best split. While doing this, it considers the split applied by Algorithm 3.2. Thus, by an inductive argument, this cannot result in a worse circuit than the one derived by Algorithm 3.2.

3.4 Global depth bound

In this section we show the global depth bound given by Theorem 3.4.6, which contains Theorem 3.1.3 as a special case with $n = 0$. We do this by using the result of Proposition 3.3.10 to get a bound on the depth depending on the input numbers instead of on the alternating inputs depending on the depth. We start by dealing with the small cases of m.

Lemma 3.4.1. *For $m \leq 3$, Algorithm 3.2 computes an optimum circuit with depth at most $\lceil \log_2(m+n) \rceil$.*

Proof. For $m = 2$ the And-Or path is an And function and therefore, Algorithm 3.2 computes an optimum circuit as observed in Lemma 3.2.7. For $m = 3$, this follows by induction. To start, there is a circuit for $(n, m) = (1, 3)$ with depth $2 = \lceil \log_2(3+1) \rceil$. For larger n the algorithm performs symmetric splits while $d \geq 6$, and (checking Table 3.3) also for $d \leq 6$. In every symmetric split, $2^{\lceil \log_2(m+n) \rceil - 1}$ inputs get split off and are built separately as a balanced binary tree with depth $\lceil \log_2(m+n) \rceil - 1$. By induction, the remaining circuit can be built with depth $\lceil \log_2(m+n) \rceil - 1$, too. Concatenating the gates yields a depth of $\lceil \log_2(m+n) \rceil$ as desired. □

Lemma 3.4.2. *Let $m = 4$ and $n \in \mathbb{Z}_{\geq 0}$. Then there is a circuit $C(s,t)$ computing $f(s,t)$ with depth at most $\log_2(m+n) + \log_2 \log_2 m$.*

Proof. We do induction on $\ell = \lfloor \log_2(n+4) \rfloor$. As a base case, consider $\ell = 2$, i.e., $n \leq 3$. We can build the circuit shown in Figure 3.8. It is of depth 3 and we can have up to 3 symmetric inputs. And indeed

$$\log_2(m+n) + \log_2 \log_2 m \geq \log_2(4) + \log_2 \log_2 4 = 3 \ .$$

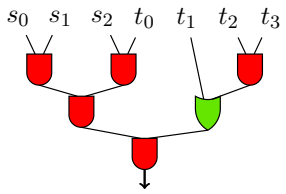


Figure 3.8: Restructured And-Or path for $m = 4$ and $n = 3$.

For the induction step, let $n \in [2^\ell - 4, 2^{\ell+1} - 5]$ for some $\ell \in \mathbb{Z}_{\geq 3}$. We have $\log_2(m+n) + \log_2 \log_2 m \geq \log_2(4 + 2^\ell - 4) + 1 = \ell + 1$, so we are looking for a circuit of depth $\ell + 1$. We do a symmetric split according to Theorem 3.2.6 for up to 2^ℓ symmetric inputs s'. Those can be built with depth ℓ by a symmetric tree by Lemma 3.2.7. The remaining $f(s \setminus s', t)$ has at most $2^{\ell+1} - 5 - 2^\ell = 2^\ell - 5$ symmetric inputs. By induction hypothesis, this implies that there exists a circuit of depth ℓ for this, too. Thus, the considered split returns a circuit of depth $\ell + 1$ as desired. □

Lemma 3.4.3. *Let $m = 5$ and $n \in \mathbb{Z}_{\geq 0}$. Then there is a circuit $C(s,t)$ computing $f(s,t)$ with depth at most $\log_2(m+n) + \log_2 \log_2 m$.*

Proof. For $n \leq 1$, we need a circuit with depth at most

$$\lfloor \log_2(m+n) + \log_2 \log_2 m \rfloor \geq \lfloor \log_2(5) + \log_2 \log_2 5 \rfloor = 3 \ .$$

Table 3.3 shows that such a circuit exists as for $m = 5$ and $d = 3$ the maximum number of symmetric inputs is 1. Otherwise, i.e., if $n \geq 2$, there is a number $\ell \in \mathbb{N}_{\geq 4}$ with

$$2^{\ell-1} - 6 \leq n \leq 2^{\ell} - 7 \ .$$

We prove by induction in ℓ that in this case there is a circuit $C(s,t)$ computing $f(s,t)$ with depth at most ℓ. For $\ell = 4$, which implies $n \in [2, 9]$, this follows from the fact that according to Table 3.3 the maximum number of symmetric inputs for $m = 5$ and $d = 4$ is 9. For the induction step from ℓ to $\ell + 1$ assume that $n \in [2^{\ell} - 6, 2^{\ell+1} - 7]$. We have to show that there is a circuit with depth $\ell + 1$ computing $f(s,t)$. We apply a symmetric split for at most 2^{ℓ} symmetric inputs s' such that the remaining $f(s \setminus s', t)$ has at most $2^{\ell} - 7$ symmetric inputs. Then the symmetric tree for s' has at most depth ℓ and (by induction) there is a circuit with depth at most ℓ computing $f(s \setminus s', t)$. This leads to a circuit with depth $\ell + 1$ computing $f(s,t)$.

Now let $n > 2$ and $\ell \in \mathbb{N}_{\geq 4}$ with $n \in [2^{\ell-1} - 6, \leq 2^{\ell} - 7]$. Since we know that there is circuit of depth at most ℓ computing $f(s,t)$, it remains to show that $\log_2(5+n) + \log_2 \log_2(5) \geq \ell$. This follows from

$$\begin{aligned}
\log_2(5+n) + \log_2 \log_2(5) &\geq \log_2(2^{\ell-1} - 1) + \log_2 \log_2(5) \\
&\overset{\ell \geq 4}{\geq} \log_2\left(\frac{7}{8} \cdot 2^{\ell-1}\right) + \log_2 \log_2(5) \\
&= \ell - 1 + \log_2\left(\frac{7}{8}\right) + \log_2 \log_2(5) \\
&\geq \ell - 1 - 0.2 + 1.2 \\
&= \ell \ . \qquad \square
\end{aligned}$$

Now that we have seen that the statement holds for small values of m, we can do the start of our induction on d, that is to prove the statement for $d \leq 6$.

Proposition 3.4.4. *Let $s = (s_0, \ldots, s_{n-1})$ and $t = (t_0, \ldots, t_{m-1})$ such that $m \geq 5$ and assume that $d := \log_2(m+n) + \log_2 \log_2 m \leq 6$. Then the optimum circuit $C(s,t)$ for the extended AND-OR path $f(s,t)$ is of depth at most d.*

Proof. First, we find that as $m \geq 5$, we have $d \geq \log_2(5) + \log_2 \log_2(5) > 3$. Thus, we distinguish the cases $\lfloor d \rfloor \in \{3, 4, 5, 6\}$. $m \geq 5$ implies that $\log_2 \log_2 m > 1.21$. In order to satisfy $\log_2(m+n) + \log_2 \log_2 m \leq 6$, this implies that $m + n < 2^{6-1.21} < 28$, so $n \leq 22$.

For $\lfloor d \rfloor = 3$, we know that $m + n < 2^{2.79} < 6.92$. Checking Table 3.3, indeed there exists a circuit of depth 3 if either $m = 5$ and $n \leq 1$ or $m = 6$ and $n = 0$.

Next, we consider $\lfloor d \rfloor = 4$. If $m \geq 10$, a more precise computation of d yields $m + n = 2^{d - \log_2 \log_2(m)} \leq 2^{5 - \log_2 \log_2(10)} < 10$, so this is impossible. Else, we know that $m + n < 2^{3.79} < 13.9$. Checking Table 3.3 at all cells $(m, n) = (m, 13 - m)$ for $m \in [5, 9]$ yields that there always exists a circuit of depth 4.

If $\lfloor d \rfloor = 5$, then $d = \log_2(m+n) + \log_2 \log_2 m$ implies $m < 16$ and we get again $m + n < 2^{6-1.21} < 28$. For $n \leq 7$, there exist circuits with depth 5 for all $m \in [5, 15]$

by Table 3.3. For $n \geq 8$ however, we have $d < 6$ only if even $m \leq 10$, in which case we find that there exists a circuit as long as $n \leq 16$. We continue by observing that $n \geq 17$ can give a depth of 5 only if $m \leq 6$. In that case however, we find that Table 3.3 yields a circuit for all $n \leq 22$ as desired.

Finally, $d = 6$ only happens for $(m, n) = (16, 0)$ in which case there even exists a circuit of depth 5. □

Next, we need the following technical statement that will be used in our induction step.

Lemma 3.4.5. *Let $m, n \in \mathbb{Z}_{\geq 0}$ and $m \geq 6$ and $\xi = 2.86$. Let*

$$\vartheta(n, m) \coloneqq \frac{\xi}{2}(m+n)\log_2 m - \xi n - 2\xi - (m-3)(\log_2(m+n) + \log_2\log_2 m - 1) \ .$$

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

Proof. We first prove the statement in the case $m \geq 7$ and later treat the case $m = 6$ individually. For $m \geq 7$ we use that $\frac{1}{\ln(2)} < 1.45$.

$$\begin{aligned}
\frac{\partial}{\partial n}\vartheta(n, m) &= \frac{\xi}{2}\log_2 m - \frac{m-3}{(m+n)\ln(2)} - \xi \\
&\geq \frac{\xi}{2}\log_2 m - \frac{m-3}{m\ln(2)} - \xi \\
&\geq \begin{cases} 0.4\xi - \frac{4}{7}\cdot 1.45 \geq 0 & m = 7 \ , \\ 0.5\xi - \frac{5}{8}\cdot 1.45 \geq 0 & m = 8 \ , \\ 0.58\xi - 1.45 \geq 0 & m \geq 9 \ . \end{cases}
\end{aligned}$$

As the partial derivative with respect to n is non-negative, it suffices to consider the case $n = 0$ and prove that $\vartheta(0, m)$ is non-negative for all $m \geq 7$. In order to do so, we also compute the partial derivative with respect to m at $n = 0$. That is,

$$\begin{aligned}
&\frac{\partial}{\partial m}\vartheta(0, m) \\
&= \frac{\partial}{\partial m}\left(\left(\frac{\xi}{2} - 1\right)m\log_2 m + 2\log_2 m - 2\xi - (m-2)(\log_2\log_2 m - 1)\right) \\
&= \left(\frac{\xi}{2} - 1\right)\left(\log_2 m + \frac{1}{\ln(2)}\right) + \frac{2}{\ln(2)m} - \log_2\log_2 m - \frac{m-3}{m\ln(2)\ln(m)} + 1 \\
&\geq \left(\frac{\xi}{2} - 1\right)\left(\log_2 m + \frac{1}{\ln(2)}\right) + \frac{2}{\ln(2)m} - \log_2\log_2 m \\
&\geq 0 \ ,
\end{aligned}$$

where we have used that $\ln(2)\ln(m) > 1$ for $m \geq 7$. Furthermore, we check the initial point $n = 0$ and $m = 7$.

$$\vartheta(0, 7) = 9.8\xi - 2\xi - 4(2.9 + 1.5 - 1) = 7.8\xi - 13.6 > 0$$

This implies that $\vartheta(n, m) \geq 0$ for all $m \geq 7$.

It remains to consider $m = 6$ separately. In this case the partial derivative is only non-negative for $n \geq 1$:

$$\begin{aligned}\frac{\partial}{\partial n}\vartheta(n,m) &= \frac{\xi}{2}\log_2 6 - \frac{3}{(6+n)\ln(2)} - \xi \\ &\geq \frac{\xi}{2}\log_2 6 - \frac{3}{7\ln(2)} - \xi \geq 0.29\xi - \frac{3}{7}\cdot 1.45 \geq 0 \ .\end{aligned}$$

In the cases $n = 0$ and $n = 1$ however, we check the values of ϑ by hand:

$$\begin{aligned}\vartheta(0,6) &= \xi(0.5\cdot 6\log_2(6) - 2) - 3(\log_2(6) + \log_2(\log_2(6)) - 1) \geq 2.86\cdot 5.75 - 9 > 0 \\ \vartheta(1,6) &= \xi(0.5\cdot 7\log_2(6) - 3) - 3(\log_2(7) + \log_2(\log_2(6)) - 1) \geq 2.86\cdot 6 - 10 > 0 \ .\end{aligned}$$

□

Theorem 3.4.6. *Let $s = (s_0, \ldots, s_{n-1})$ and $t = (t_0, \ldots, t_{m-1})$ such that $m + n \geq 1$. There exists a polynomial time algorithm that computes a circuit $C(s,t)$ for the extended AND-OR path of depth*

$$\operatorname{depth}(C(s,t)) \begin{cases} = \lceil \log_2(m+n) \rceil & \text{for } m \leq 3 \text{ and} \\ \leq \log_2(m+n) + \log_2\log_2 m & \text{for } m \geq 4. \end{cases}$$

Proof. For $m \leq 3$, Algorithm 3.2 computes an optimum circuit of the desired depth by Lemma 3.4.1. If $m = 4$ or $m = 5$, the result follows from Lemma 3.4.2 or Lemma 3.4.3. So let $m \geq 6$. Proposition 3.4.4 shows that there exists such a circuit whenever $\log_2(m+n) + \log_2\log_2 m \leq 6$. As Algorithm 3.2 computes an optimum circuit in these cases, the theorem is clearly true for $d \leq 6$.

For $d \geq 7$ we define $\tilde{d} := \lfloor \log_2(m+n) + \log_2\log_2 m \rfloor$, and we prove that $m \leq \mu(\tilde{d}, n)$. Note that we have $n < 2^{\tilde{d}} - 2$, so $\mu(\tilde{d}, n) = \xi_d \frac{2^{\tilde{d}} - n - 2}{\tilde{d}} + 3$.

As $\mu(d,n)$ is monotonically increasing in d for fixed n, it suffices to show that $m \leq \mu(\hat{d}, n)$ where $\hat{d} := \log_2(m+n) + \log_2\log_2 m - 1$. Then Proposition 3.3.10 implies a circuit for $f(s,t)$. Plugging in $\mu(\hat{d}, n)$, using the lower bound $\xi := 2.86 \leq \xi_d$ for all $d \geq 7$, and rearranging, we get

$$\begin{aligned} & m \leq \xi \frac{2^{\log_2(m+n)+\log_2\log_2 m - 1} - n - 2}{\log_2(m+n) + \log_2\log_2 m - 1} + 3 \\ \Leftrightarrow \quad & m \leq \xi \frac{\frac{1}{2}(m+n)\log_2 m - n - 2}{\log_2(m+n) + \log_2\log_2 m - 1} + 3 \\ \Leftrightarrow \quad & 0 \leq \frac{\xi}{2}(m+n)\log_2 m - \xi n - 2\xi - (m-3)(\log_2(m+n) + \log_2\log_2 m - 1) \ . \end{aligned} \tag{3.15}$$

This inequality has already been proven in Lemma 3.4.5. □

3.5 Asymptotic depth bound

In Section 3.4 we established a global depth bound for our algorithm. This bound is tight for $m = 4$. Thus, instead of improving on this bound for all m, we analyze the asymptotic behavior of the depth of our circuits. We show an asymptotic lower bound together with a convergence rate to this asymptotic lower bound.

Theorem 3.5.1. *Let $s = (s_0, \ldots, s_{n-1})$ and $t = (t_0, \ldots, t_{m-1})$ such that $m \geq 6$. Then there exists a polynomial time algorithm that computes a circuit $C(s,t)$ for the extended And-Or path $f(s,t)$ of depth*

$$\begin{aligned}\operatorname{depth}(C(s,t)) \quad &\leq \quad \log_2(m+n) - \log_2\left(4 - \frac{3}{(\log_2(m+n) + \log_2\log_2 m - 2)^{0.5}}\right) \\ &\quad + \log_2\big(\log_2(m+n) + \log_2\log_2 m\big) + 1 \ .\end{aligned}$$

Proof. For $m, n \in \mathbb{Z}_{\geq 0}$ set

$$\begin{aligned}\gamma(m,n) \quad &= \quad \log_2\left(4 - \frac{3}{(\log_2(m+n) + \log_2\log_2 m - 2)^{0.5}}\right) + \log_2\log_2 m \\ &\quad - \log_2\big(\log_2(m+n) + \log_2\log_2 m\big) \ .\end{aligned}$$

Then $\gamma(m,n) \leq 2$ as the first summand is at most two and the third has a larger absolute value than the second. We need to show that we can compute a circuit $C(s,t)$ for the extended And-Or path $f(s,t)$ of depth

$$\operatorname{depth}(C(s,t)) \quad \leq \quad \log_2(m+n) + \log_2\log_2 m - \gamma(m,n) + 1 \ .$$

If $\gamma(m,n) \leq 1$, we can use the circuit from Theorem 3.4.6. Hence, we assume in the following that $\gamma(m,n) \geq 1$. We proceed similarly to Theorem 3.4.6 and define $\hat{d} \coloneqq \lfloor \log_2(m+n) + \log_2\log_2 m - \gamma(m,n) + 1 \rfloor$, and we prove that $m \leq \mu(\hat{d}, n)$.

Claim. $n \leq 2^{\hat{d}} - 3$.

Proof of claim. To prove the claim, assume that $n \geq 2^{\hat{d}} - 2$. If in addition $m \geq 16$, we get

$$\begin{aligned} n \quad &\geq \quad 2^{\hat{d}} - 2 \\ &\geq \quad \frac{(m+n)\log_2 m}{2^{\gamma(m,n)}} - 2 \\ &\overset{\gamma(m,n)\leq 2}{\geq} \quad \frac{(m+n)\log_2 m}{4} - 2 \\ &\overset{m\geq 16}{\geq} \quad \frac{(16+n)\log_2 16}{4} - 2 \\ &= \quad n + 14 \ , \end{aligned}$$

which is a contradiction. If $m \leq 15$, then

$$\begin{aligned}\gamma(m,n) &\leq 2 + \log_2\log_2 m - \log_2\big(\log_2(m+n) + \log_2\log_2 m\big) \\ &\leq 2 + \log_2\log_2 15 - \log_2\big(\log_2(15+n) + \log_2\log_2 15\big) \\ &\leq 2 + \log_2\log_2 15 - \log_2(\log_2 15 + \log_2\log_2 15) \\ &\leq 2 - 0.58 = 1.42 \end{aligned} \tag{3.16}$$

and thus

$$\begin{aligned} n \quad &\geq \quad 2^{\hat{d}} - 2 \\ &\geq \quad \frac{(m+n)\log_2 m}{2^{\gamma(m,n)}} - 2 \\ &\overset{\gamma(m,n)\leq 1.42}{\geq} \quad \frac{(m+n)\log_2 m}{2.68} - 2 \\ &\overset{m\geq 6}{\geq} \quad \frac{(6+n)\log_2 6}{2.68} - 2 \geq 0.96n + 3.7 \ , \end{aligned}$$

which implies $0.04n \geq 3.7$ and hence $n \geq 93$.

Therefore, Equation (3.16) even yields

$$\gamma(m,n) \leq 2 + \log_2 \log_2 15 - \log_2(\log_2(99) + \log_2 \log_2 6) \leq 2 - 1.03 < 1 \ ,$$

which contradicts $\gamma(m,n) \geq 1$. This proves the claim. □

Hence, we can assume that $n < 2^{\hat{d}} - 2$ and thus $\mu(\hat{d},n) = \xi_{\hat{d}} \frac{2^{\hat{d}}-n-2}{\hat{d}} + 3$. As $x \mapsto \frac{2^x-n-2}{x}$ is monotonically increasing and $\hat{d} \geq \log_2(m+n) + \log_2 \log_2 m - \gamma(m,n)$, it suffices to show

$$m \ \leq \ \xi_{\hat{d}} \frac{2^{\log_2(m+n)+\log_2 \log_2 m - \gamma(m,n)} - n - 2}{\log_2(m+n) + \log_2 \log_2 m - \gamma(m,n)} + 3 \tag{3.17}$$

which is equivalent to

$$m \leq \xi_{\hat{d}} \frac{2^{-\gamma(m,n)}(m+n)\log_2 m - n - 2}{\log_2(m+n) + \log_2 \log_2 m - \gamma(m,n)} + 3$$

and

$$\begin{aligned} 0 \leq \xi_{\hat{d}} 2^{-\gamma(m,n)}(m+n)\log_2 m - \xi_{\hat{d}} n - 2\xi_{\hat{d}} \\ - (m-3)\Big(\log_2(m+n) + \log_2 \log_2 m - \gamma(m,n)\Big) \ . \end{aligned}$$

In addition, we have $\hat{d} \geq \log_2(m+n) + \log_2 \log_2 m - 2$, and thus

$$\xi_{\hat{d}} \geq 4 - \frac{3}{(\log_2(m+n) + \log_2 \log_2 m - 2)^{0.5}} \ .$$

Using this and the definition of $\gamma(m,n)$, it is sufficient to show that

$$\begin{aligned} 0 \leq (m+n)(\log_2(m+n) + \log_2 \log_2 m) - \xi_{\hat{d}} n - 2\xi_{\hat{d}} \\ - (m-3)\Big(\log_2(m+n) + \log_2 \log_2 m - \gamma(m,n)\Big) \ , \end{aligned}$$

which in turn is equivalent to

$$\begin{aligned} 0 \leq n(\log_2(m+n) + \log_2 \log_2 m) - \xi_{\hat{d}} n - 2\xi_{\hat{d}} \\ + (m-3)\gamma(m,n) + 3(\log_2(m+n) + \log_2 \log_2 m) \ . \end{aligned} \tag{3.18}$$

Now, if $n \geq 1$ then $\log_2(m+n) + \log_2 \log_2 m \geq \log_2 7 + \log_2 \log_2 6 > 4.1 > \xi_{\hat{d}}$, so

$$n(\log_2(m+n) + \log_2 \log_2 m) - \xi_{\hat{d}} n \geq 0 \ ,$$

which is also obviously true for $n = 0$. Moreover, we have $(m-3)\gamma(m,n) \geq 3$ and $3(\log_2(m+n) + \log_2 \log_2 m) \geq 3(\log_2(6) + \log_2 \log_2 6) \geq 11.8$, so we have

$$-2\xi_{\hat{d}} + (m-3)\gamma(m,n) + 3(\log_2(m+n) + \log_2 \log_2 m) \geq 0 \ .$$

This shows (3.18) and thus (3.17). □

Corollary 3.5.2. *For all $m \in \mathbb{Z}_{\geq 6}$, and $t = (t_0, \ldots, t_{m-1})$, there exists a polynomial time algorithm that computes a circuit $C(t)$ for the AND-OR path $g(t)$ of depth*

$$\begin{aligned} \mathrm{depth}(C(t)) \leq \ & \log_2 m - \log_2\left(4 - \frac{3}{(\log_2 m + \log_2 \log_2 m - 2)^{0.5}}\right) \\ & + \log_2(\log_2 m + \log_2 \log_2 m) + 1 \ . \end{aligned}$$

Remark 3.5.3. As

$$\lim_{m\to\infty} \log_2\left(4 - \frac{3}{(\log_2 m + \log_2\log_2 m - 2)^{0.5}}\right) = 2$$

and

$$\lim_{m\to\infty} \log_2(\log_2 m + \log_2\log_2 m) - \log_2\log_2 m = 0 \ ,$$

this upper bound asymptotically behaves like $\log_2 m + \log_2\log_2 m - 1$.

Remark 3.5.4. For $m \geq 73$, we have

$$-\log_2\left(4 - \frac{3}{(\log_2 m + \log_2\log_2 m - 2)^{0.5}}\right) + \log_2(\log_2 m + \log_2\log_2 m) + 1 \leq \log_2\log_2 m \ ,$$

so in this range the upper bound in the above corollary is better than the bound of $\log_2 m + \log_2\log_2 m$ that follows from Theorem 3.4.6 for $n = 0$.

3.6 Global size bound

So far we have analyzed Algorithm 3.2 with respect to the achievable depth. In this section, we improve the algorithm such that it achieves a linear size And-Or path. To that extend, we proceed similarly to Brenner and Silvanus [BS24].

The reason why the size with the naive approach of Algorithm 3.2 is superlinear is that after every alternating split, we get a prefix of And or Or gates. Recursive splitting gives many such prefixes whose size we need to bound in total. Our goal is to construct a solid base of And and Or trees from which the gates can be reused. Whenever we do a symmetric split, we carefully choose those inputs of the prefix for which we can reuse a maximum number of gates. The solid basis of And tres and Or trees is arranged in so-called leftist circuits. These special circuits have been introduced by Brenner and Silvanus [BS24] for their algorithm and work for our algorithm just in the same way.

3.6.1 Leftist Circuits

We follow the definitions and results of Brenner and Silvanus [BS24] and will use some results they derived about leftist circuits.

Definition 3.6.1. Let $x_0, \ldots, x_{n-1}$ be inputs of a circuit. We call a subset $K \subseteq \{x_0, \ldots, x_{n-1}\}$ **consecutive** if $K = \{x_i \mid a \leq i < b\}$ for integers $0 \leq a \leq b < n$.

Definition 3.6.2. Let S be a circuit over inputs $x_0, \ldots, x_{n-1}$. The circuit S is called **ordered** if the underlying undirected graph is acyclic and for every vertex v the set $\mathcal{I}_v$ (see Notation 2.1.10) is a consecutive set.

Definition 3.6.3. Let $n \in \mathbb{Z}_{\geq 1}$ and let $x_0, \ldots, x_{n-1}$ be n inputs. Let $\circ \in \{\text{And2}, \text{Or2}\}$ be a symmetric gate. Moreover, let $\ell \in \mathbb{Z}_{\geq 0}$ be chosen such that $2^\ell \leq n < 2^{\ell+1}$. The **leftist circuit** over $\circ$ on inputs $x_0, \ldots, x_{n-1}$ is recursively defined as follows. It contains the unique ordered balanced binary tree of $\circ$-gates on the inputs $x_0, \ldots, x_{2^\ell-1}$. Additionally, it consists of the leftist circuit on $x_{2^\ell}, \ldots, x_{n-1}$.

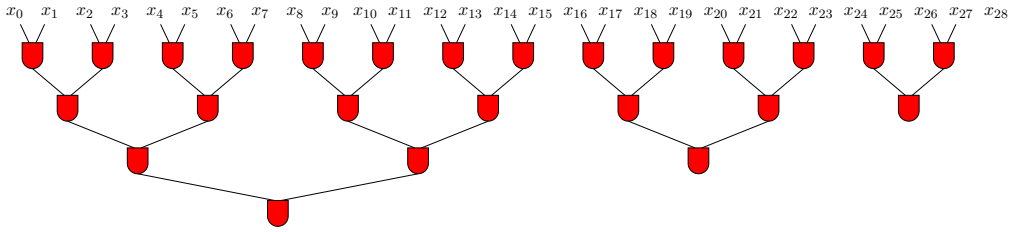


Figure 3.9: Leftist circuit over AND gates on 29 inputs.

An example of a leftist circuit can be seen in Figure 3.9. To achieve linear size extended AND-OR paths for $f(s,t)$, we create two leftist circuits S_0 and S_1: circuit S_0 is a leftist AND circuit on s and on the inputs of t with even index, and S_1 a leftist OR circuit on the inputs of t with odd index. This is a reasonable choice because whenever we apply an alternating split the newly obtained prefix contains every second input. Moreover, it is a prefix of AND gates if and only if the inputs are part of s or their indices in t are even. So whenever we need to build a symmetric function due to an alternating split, we can reuse gates from one of the leftist circuits. This symmetric function will always depend on a consecutive set of inputs. Computing the symmetric function then reduces to combining the reusable intermediate results from the leftist circuit. To be more precise about the reusable intermediate results, we define boundary vertices as follows.

Definition 3.6.4. Let S be a leftist circuit and K be a subset of $\mathcal{I}(S)$. The **boundary vertices** $B(K,S)$ of S with respect to K are those vertices $v \in \mathcal{V}(S)$ for which we have $\mathcal{I}_v \subseteq K$ but for all $w \in \delta^+(v)$ we have $\mathcal{I}_w \not\subseteq K$.

A vertex v is left of a vertex w if for every $x_i \in \mathcal{I}_v$ and $x_j \in \mathcal{I}_w$ we have $i < j$. Note that this relation induces a total order on the boundary vertices as their sets $(\mathcal{I}_v)_{v \in K}$ are pairwise disjoint.

If we need an AND circuit on a subset K of the inputs, we just need to combine the boundary vertices with respect to K which will usually be much cheaper than building a circuit from scratch. To even improve on this, we introduce triangular sets. Those sets will be particularly cheap to build because their number of boundary vertices is at most logarithmic in the set size.

Definition 3.6.5. Let S be a leftist circuit and for a subcircuit $T \subseteq S$ let $\text{depth}(T)$ be its depth, i.e., the length of the longest path in T. Let $K \subseteq \mathcal{I}(S)$ and let $B := B(K,S)$ be the set of boundary vertices. The **boundary tree sequence** for K is the sequence $(T_0, \ldots, T_{|B|-1})$ of binary subtrees of S such that each T_i is the cone[1] of a boundary vertex $v_i \in B$ and such that the v_i are sorted from left to right, i.e., that v_i is left of v_j if and only if $0 \leq i < j < |B|$.

K is called **triangular** if there exists a $0 \leq j \leq |B| - 1$ such that the inputs of $T_0, \ldots, T_j$ are consecutive, the inputs of $T_{j+1}, \ldots, T_{|B|-1}$ are consecutive, and $\text{depth}(T_i) < \text{depth}(T_{i+1})$ for $0 \leq i < j$ and $\text{depth}(T_i) > \text{depth}(T_{i+1})$ for $j < i < |B| - 2$.

An example for a triangular set is shown in Figure 3.10. Whenever we apply a symmetric split as in line 7 of Algorithm 3.4, we maintain two triangular sets s'

[1] A vertex v is in the cone of w if and only if there is a v-w-path.

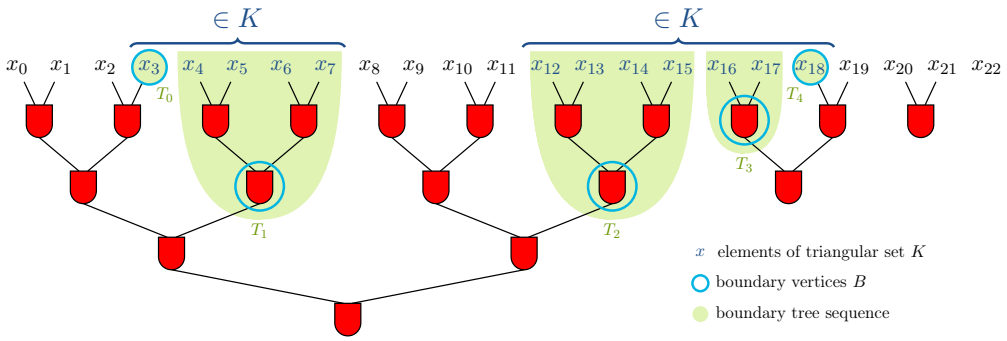


Figure 3.10: Example for a triangular set K. The And gates show the leftist circuit. The set K itself is shown in dark blue. The boundary vertices for K are the ones highlighted in light blue. The boundary tree sequence is shown in green. The set K is triangular because for $j = 1$, the inputs of T_0 and T_1 are consecutive, the inputs of T_2, T_3 and T_4 are also consecutive, and the depth of the boundary trees is first increasing and then decreasing.

and $s \setminus s'$. This is done using Algorithm 3.3 which has been introduced by Brenner and Silvanus [BS24]. They show that this algorithm is able to choose a subset out of a triangular set such that both the subset and its complement are also triangular. For this, it proceeds as follows: given a set N, it is supposed to choose a subset K of size 2^{d-1}. If the boundary tree sequence of N contains a tree of depth $d-1$, it just takes the corresponding inputs. Else, it searches for a pair of trees with the same depth, takes the largest such trees and returns all the inputs contained in these trees or in trees in between them. We will see a consequence of this result in Lemma 3.6.9. Before, we consider the solution guarantee derived for the size of the boundary vertices of such a decomposition.

Algorithm 3.3: 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 $2^{d-1} \leq |N| < 2^d$.

Output: a set $K \subseteq N$.

Let $T_0, \dots, T_{|B(N,S)|-1}$ denote the boundary tree sequence for N w.r.t. S
if *There is a* $j \in \{0, \dots, |B(N,S)| - 1\}$ *with* $\operatorname{depth}(T_j) = d - 1$ **then**
  return $\mathcal{I}(T_j)$
else
  $D \leftarrow \max\{d' \in \mathbb{N} \mid \exists\, j_0 < j_1$ with $\operatorname{depth}(T_{j_0}) = \operatorname{depth}(T_{j_1}) = d'\}$.
  Choose $j_0 < j_1$ such that $\operatorname{depth}(T_{j_0}) = \operatorname{depth}(T_{j_1}) = D$.
  return $\bigcup_{j=j_0}^{j_1} \mathcal{I}(T_j)$

Notation 3.6.6. We define the function $\rho : \mathbb{N} \to \mathbb{R}_{\geq 0}$ by

$$\rho(n) = \begin{cases} n & \text{if } n \leq 2 \\ \lfloor 2 \log_2(n-1) \rfloor & \text{if } n \geq 3 \end{cases}. \tag{3.19}$$

Theorem 3.6.7 ([BS24], Theorem 3.34)**.** *Let S be a leftist circuit. Consider a triangular subset $N \subseteq \mathcal{I}(S)$ with $|N| = n \geq 1$. Let $k \in \mathbb{Z}_{\geq 1}$ be the unique power of two with $k \leq n < 2k$. Let $K \subseteq N$ be the output of Algorithm 3.3 given S and N, and let $B \coloneqq B(K, S)$ be the boundary vertices of K with respect to S. Then, the following statements are fulfilled.*

(i) If $n - k \geq 2$, we have $|B| + \lfloor 2\log_2(n-k-1) \rfloor \leq \lfloor 2\log_2(n-1) \rfloor$.

(ii) For $n \geq 16$, we have $|B| + \rho(n-k) \leq \rho(n)$.

3.6.2 Improved algorithm for linear size AND-OR paths

In order to linearize the number of gates used by our circuits, we apply two modifications to our algorithm. The modified version is shown in Algorithm 3.4, highlighting the new parts of the algorithm. We use for $p \in \mathbb{N}$ the notation $\mathrm{flodd}(p)$ to denote the largest odd integer not larger than p.

Algorithm 3.4: Depth optimization with improved size for extended AND-OR paths.

Input: $n, m \in \mathbb{Z}_{\geq 0}$, a set of symmetric inputs $s = (s_0, \ldots, s_{n-1})$, a set of alternating inputs $t = (t_0, \ldots, t_{m-1})$ and a depth threshold d_0.
Output: a circuit $C(s,t)$ computing $f(s,t)$.

1 **if** $m \leq 2$ **then**
2 **return** *Optimum circuit for* $\textsc{And}(s_0, \ldots, s_{n-1}, t_0, \ldots, t_{m-1})$
3 $d \leftarrow \min\{d \mid m \leq \mu(d,n)\}$
4 **if** $d \leq d_0$ **then**
5 **return** *Optimum circuit for* $f(s,t)$
6 **if** $n \geq 2^{d-1}$ **then**
7 Choose $s' \subseteq s$ such that $|s'| = 2^{d-1}$.
8 Compute an optimum circuit S for $\textsc{And}(s')$.
9 **return** $S \wedge C(s \setminus s', t)$
10 **else**
11 **if** $m \leq \mu(d-1, 0)$ **then**
12 Compute an optimum circuit S for $\textsc{And}((s_0, \ldots, s_{n-1}))$
13 **return** $S \wedge C((), t)$
14 $k \leftarrow \min\left\{ \max\{k \mid k \text{ odd}, k \leq \mu(d-1, n)\}, \ \mathrm{flodd}\left(\frac{2}{3}m\right) + 2 \right\}$.
15 Let $t' \leftarrow (t_0, \ldots, t_{k-1})$ and $t'' \leftarrow t \setminus t'$.
16 **return** $C(s,t') \wedge \left(C(\widehat{t'}, t'')\right)^*$

These bounds on the depth as stated in Theorem 3.5.1 and Corollary 3.5.2 are still valid, which follows from the following theorem.

Proposition 3.6.8. *For any instance, Algorithm 3.4 computes a circuit whose depth is as most as big as the depth of the circuit computed by Algorithm 3.2.*

Proof. Both algorithms compute the same depth d in line 3, and we have to check that if $d > d_0$ (the case $d \leq d_0$ being trivial) also Algorithm 3.4 computes a circuit with deph at most d. If $n \geq 2^{d-1}$, this follows by induction. But the same is true if

$n < 2^{d-1}$ and $m \leq \mu(d-1,0)$ because both S and $C((),t)$ will be built with depth at most $d-1$.

Finally, we only need to show that choosing $k \leq \operatorname{flodd}\left(\frac{2}{3}m\right)+2$ does not increase the depth of the computed circuit.

Assume that Algorithm 3.2 performs a split such that $k > \operatorname{flodd}\left(\frac{2}{3}m\right)+2$ to get a circuit of depth d. Then, Algorithm 3.4 splits along $k' = \operatorname{flodd}\left(\frac{2}{3}m\right)+2$. We claim that indeed, this split does not achieve a larger depth than d. Let the primary split with k alternating inputs subdivide t into t' and t'', and let the alternative split with k' alternating inputs subdivide t into r' and r''. The first subcircuit of the alternative split computes $f(s,r')$ where $|r'| = k'$. The depth of this circuit will be at most the depth of $f(s,t')$ where $|t'| = k$ as $k > k'$. So the depth of the first subcircuit is at most $d-1$ and it remains to show a similar statement for the second part.

When applying the alternative split, the second circuit computes $f(\widehat{r'}, r'')$. We claim that it is possible to compute a circuit for $m' = \lfloor \frac{1}{3}m + 0.5 \rfloor$ alternating inputs with depth $d-2$. We know that it is possible to compute $f(s,t')$ using a circuit of depth $d-1$. This can either be achieved by a symmetric or by an alternating split. In either split, at least one subcircuit must contain at least m' alternating inputs because $k \geq 2m'$. This subcircuit is constructed with depth $d-2$ by the algorithm. But then, this is also true for an instance of m' alternating inputs without any possible additional symmetric inputs. We know that $|\widehat{r'}| = \frac{k'-1}{2} = \lfloor \frac{1}{3}m + 0.5 \rfloor$ and $|r''| = m - k' < \frac{1}{3}m$. One possible split to compute $f(\widehat{r'}, r'')$ is the symmetric split according to Theorem 3.2.6 splitting off all the prefix. For this split, it remains to build a functions with less than $\frac{1}{3}m$ symmetric inputs and a function with m' alternating inputs with depth $d-2$. We have shown above that the second is possible, and the first follows from the second. □

We use this improved algorithm in combination with the leftist circuits introduced in the previous subsection to derive our final algorithm Algorithm 3.5. We will in the following analyze this algorithm in more detail.

Algorithm 3.5: Depth optimization with linear size for extended And-Or paths.

Input: $n, m \in \mathbb{Z}_{\geq 0}$, a set of symmetric inputs $s = (s_0, \ldots, s_{n-1})$, a set of alternating inputs $t = (t_0, \ldots, t_{m-1})$.

Output: a circuit $C(s,t)$ computing $f(s,t)$.

1 Construct a leftist And circuit on inputs $s_0, \ldots, s_{n-1}, t_0, t_2, \ldots, t_{2\lfloor \frac{m-1}{2} \rfloor}$.

2 Construct a leftist Or circuit on inputs $t_1, t_3, \ldots, t_{2\lfloor \frac{m}{2} \rfloor - 1}$.

3 Apply Algorithm 3.4, using Algorithm 3.3 as a subroutine to determine s' in line 7.

Brenner and Silvanus [BS24] have already shown that this algorithm indeed maintains a triangular subset for all subinstances arising in recursive calls. The following lemma implies that we can assume that whenever we compute a subcircuit, our prefix is a triangular subset.

Lemma 3.6.9 ([BS24], Lemma 3.39)**.** *Assume the scenario of calling Algorithm 3.5, i.e., constructing S_0 and S_1 as given and then calling Algorithm 3.4. Consider*

a recursive call of Algorithm 3.4 to compute $f(s',t')$ or $f^*(s',t')$ with symmetric inputs $s' = (s_0,\dots,s_{n-1})$ and $t' = (t_0,\dots,t_{m-1})$. Then the sets s' and $s' + t_0$ are both triangular with respect to S_0 or to S_1, respectively.

This is important to be able to use Theorem 3.6.7 for every subinstance, which is needed to estimate the number of gates we need to add additionally to the leftist circuits. We do not compute this number explicitly but instead give an upper bound.

3.6.3 Bounding the number of additional gates

In the first iteration of our algorithm, we can always ensure that $s_0 \wedge s_1$ is a result contained in the leftist tree because of the leftist property. After some recursive calls to the algorithm however, we can no longer ensure that the first two inputs of the prefix of our current instance are combined by a vertex in the leftist circuit. For example, it might happen that we need to build a symmetric function on two inputs twice for different input pairs and once, it requires a gate additional to the leftist circuit and once it does not. Thus, note that the number of additional gates for a subinstance is not an independent property but depends on the main instance and on its shift towards the leftist circuit. Hence, we cannot always just compute the additional size that was necessary if we had built the leftist circuit just for this instance. Instead, we estimate the maximum number of additional gates a circuit with n symmetric and m alternating inputs needs over all possible arising inputs. Then we can ensure that if we have to solve a subinstance with n symmetric and m alternating inputs then $\mathrm{AddSize}(n,m)$ additional gates suffice.

Definition 3.6.10 (additional gates)**.** Let $n, m \in \mathbb{Z}_{\geq 0}$ be two integers. For a subcircuit C which is computed at a recursive step of Algorithm 3.4 within Algorithm 3.5 with n symmetric and m alternating inputs we set $\mathrm{AddSize}(C)$ to be the number of gates this circuit contains that are not part of any leftist circuit. Now for $n, m \in \mathbb{Z}_{\geq 0}$ we set the **additional size** $\mathrm{AddSize}(n,m)$ to be the maximum $\mathrm{AddSize}(C)$ over all possible subcircuits C of depth d computed by Algorithm 3.5 over all input tuples.

In the following we prove a bound on $\mathrm{AddSize}(n,m)$ and write this bound as

$$\psi(d,n,m) \coloneqq \beta_d m + \rho(n) - 1$$

where $\rho(n)$ is defined as in Equation (3.19). By induction, we show that if Algorithm 3.4 builds a circuit for n symmetric and m alternating inputs of depth d, then

$$\mathrm{AddSize}(n,m) \leq \psi(d,n,m) \ . \tag{3.20}$$

The numbers β_d used in this definition need to be increasing in d to allow an induction step by alternating splits. We set

$$\beta_d \coloneqq 1.3 + \sum_{i=5}^{d} \frac{(i-1)}{\xi_{i-1} 2^{i-2}} \left(i + \log_2\left(\frac{\xi_{i-1}}{3(i-1)} \right) \right) \ .$$

Inductive steps

We start with a very simple observation.

Observation 3.6.11. For every split of a circuit C into circuits C_1 and C_2 we have

$$\mathrm{AddSize}(C) \leq \mathrm{AddSize}(C_1) + \mathrm{AddSize}(C_2) + 1 \ .$$

We prove the induction step separately for the three possible splits and make use Observation 3.6.11 several times.

Lemma 3.6.12 (Symmetric split size)**.** *Let $n, m \in \mathbb{Z}_{\geq 0}$ such that Algorithm 3.4 returns a circuit of depth $d \geq 5$ by a symmetric split in line 9, i.e., such that $n \geq 2^{d-1}$. Assume that* $\text{AddSize}(n', m') \leq \psi(d-1, n', m')$ *for all $n', m' \in \mathbb{Z}_{\geq 0}$ for which Algorithm 3.4 returns a circuit of depth $d-1$. Then* $\text{AddSize}(n, m) \leq \psi(d, n, m)$.

Proof. Let (s, t) be the input variables. We split $C(s,t) = \textsc{And}(s') \wedge C(s \setminus s', t)$ where s' is a subset of s of size 2^{d-1}. We construct s' using Theorem 3.6.7. Due to the split, we know that $n \geq 2^{d-1} \geq 16$ and can thus use Item (ii). From this, we get that $|B(s', S)| + \rho(n - 2^{d-1}) \leq \rho(n)$. We need at most $|B(s', S)| - 1$ additional gates for the And circuit.

$$\begin{aligned}
\text{AddSize}(n, m) &\leq \psi(d-1, n-2^{d-1}, m) + |B(s', S)| - 1 + 1 \\
&= \beta_{d-1} m + \rho(n - 2^{d-1}) - 1 + |B(s', S)| \\
&\leq \beta_{d-1} m + \rho(n) - 1 \\
&\leq \psi(d, n, m) \ .
\end{aligned}$$

□

Lemma 3.6.13 (Prefix split size)**.** *Let $n, m \in \mathbb{Z}_{\geq 0}$ such that Algorithm 3.4 returns a circuit of depth $d \geq 5$ by a symmetric split in line 13, i.e., such that $m \leq \mu(d-1, 0)$. Assume that* $\text{AddSize}(n', m') \leq \psi(d-1, n', m')$ *for all $n', m' \in \mathbb{Z}_{\geq 0}$ for which Algorithm 3.4 returns a circuit of depth $d-1$. Then* $\text{AddSize}(n, m) \leq \psi(d, n, m)$.

Proof. When splitting off all the prefix, the circuit can be built with the additional sizes of the two subcircuits plus an additional gate for the concatenation. Thus, by Observation 3.6.11

$$\begin{aligned}
\text{AddSize}(n, m) &\leq \text{AddSize}(n, 0) + \text{AddSize}(0, m) + 1 \\
&\leq \psi(d-1, n, 0) + \psi(d-1, 0, m) + 1 \\
&= \beta_{d-1} m - 1 + \rho(n) - 1 + 1 \\
&\leq \psi(d, n, m) \ .
\end{aligned}$$

□

Lemma 3.6.14 (Alternating split size)**.** *Let $n, m \in \mathbb{Z}_{\geq 0}$ such that Algorithm 3.4 returns a circuit by an alternating split with depth $d \geq 5$ in line 16, i.e., such that $n < 2^{d-1}$ and $m > \mu(d-1, 0)$. Assume that* $\text{AddSize}(n', m') \leq \psi(d-1, n', m')$ *for all $n', m' \in \mathbb{Z}_{\geq 0}$ for which Algorithm 3.4 returns a circuit of depth $d-1$. Then* $\text{AddSize}(n, m) \leq \psi(d, n, m)$.

Proof. In Algorithm 3.4, k is chosen such that $k \leq \frac{2m}{3} + 2$, and an alternating split is performed. Thus, because of Observation 3.6.11 we need to prove

$$\psi(d, n, m) \geq \psi(d-1, n, k) + \psi\left(d-1, \frac{k-1}{2}, m-k\right) + 1 \ . \tag{3.21}$$

We calculate

$$\begin{aligned}
&\psi(d,n,m) - \psi(d-1,n,k) - \psi\left(d-1, \frac{k-1}{2}, m-k\right) - 1 \\
&= \beta_d m + \rho(n) - 1 - \beta_{d-1}k - \rho(n) + 1 - \beta_{d-1}(m-k) - \rho\left(\frac{k-1}{2}\right) \\
&= (\beta_d - \beta_{d-1})m - \rho\left(\frac{k-1}{2}\right) \\
&\geq \frac{(d-1)}{\xi_{d-1}2^{d-2}}\left(d + \log_2\left(\frac{\xi_{d-1}}{3(d-1)}\right)\right)m - \rho\left(\frac{k-1}{2}\right) \ .
\end{aligned}$$

We have to show that this term in nonnegative. We use $m > \mu(d-1,0)$ to bound the left part and get

$$\begin{aligned}
\frac{(d-1)}{\xi_{d-1}2^{d-2}}\left(d + \log_2\left(\frac{\xi_{d-1}}{3(d-1)}\right)\right)m &> \frac{(d-1)}{\xi_{d-1}2^{d-2}}\left(d + \log_2\left(\frac{\xi_{d-1}}{3(d-1)}\right)\right)\xi_{d-1}\frac{2^{d-1}}{d-1} \\
&= 2\left(d + \log_2\left(\frac{\xi_{d-1}}{3(d-1)}\right)\right) \\
&\geq 2\left(d + \log_2\left(\frac{2}{3(d-1)}\right)\right) \\
&\overset{d\geq 5}{\geq} 4 \ ,
\end{aligned}$$

which proves $\frac{(d-1)}{\xi_{d-1}2^{d-2}}\left(d + \log_2\left(\frac{\xi_{d-1}}{3(d-1)}\right)\right)m - \rho\left(\frac{k-1}{2}\right) \geq 0$ if $k \leq 5$ and thus $\rho\left(\frac{k-1}{2}\right) = \frac{k-1}{2}$. If $k > 5$ then

$$\rho\left(\frac{k-1}{2}\right) = \left\lfloor 2\log_2\left(\frac{k-1}{2} - 1\right)\right\rfloor \leq 2\log_2\left(\frac{k-1}{2}\right) \ .$$

We get

$$\begin{aligned}
\frac{2}{m}\log_2\left(\frac{k-1}{2}\right) &\leq \frac{2}{m}\log_2\left(\frac{\frac{2m}{3}+1}{2}\right) \\
&\leq \frac{2(d-1)}{\xi_{d-1}2^{d-1}}\log_2\left(\xi_{d-1}\frac{2^{d-1}}{3(d-1)} + \frac{1}{2}\right) \\
&\overset{d\geq 5}{\leq} \frac{2(d-1)}{\xi_{d-1}2^{d-1}}\log_2\left(\xi_{d-1}\frac{2\cdot 2^{d-1}}{3(d-1)}\right) \\
&= \frac{(d-1)}{\xi_{d-1}2^{d-2}}\left(d + \log_2\left(\frac{\xi_{d-1}}{3(d-1)}\right)\right) \ .
\end{aligned}$$

In the second step, we used that $\frac{1}{m}\log_2\left(\frac{2m+3}{6}\right)$ is strictly decreasing for $m \geq 5$. □

Small circuits

The previous three lemmas conclude the cases we need for the induction step. It remains to do a good start for the induction. For this, we need an additional lemma on the additional size that are stronger than Observation 3.6.11.

Lemma 3.6.15. *Let n, m be such that Algorithm 3.4 applies a symmetric split with prefix length k and $n \geq 16$ or $n - k \geq 2$. Then*

$$\mathrm{AddSize}(n, m) \leq \mathrm{AddSize}(n - k, m) + \rho(n) - \rho(n - k) \ .$$

Proof. This lemma mainly follows from Theorem 3.6.7. This theorem implies that for every circuit C with n symmetric inputs for which $n - k \geq 2$ or $n \geq 16$ we have $|B| \leq \rho(n) - \rho(n - k)$. Here, the set B is the set of boundary vertices of the triangular set K of size k selected by Algorithm 3.3 from the n symmetric inputs. Thus, its additional size $\mathrm{AddSize}(C) \leq \mathrm{AddSize}(n - k, m) + \rho(n) - \rho(n - k)$ because it takes at most $\rho(n) - \rho(n - k) - 1$ additional vertices for the k symmetric inputs. As $\mathrm{AddSize}(n, m) = \min_C \mathrm{AddSize}(C)$, this also implies the lemma. □

Apart from this we need the following lemma that is due to Brenner and Silvanus [BS24] to determine precise additional sizes for $m = 0$, i.e., for symmetric functions.

Lemma 3.6.16 ([BS24], Lemma 3.23)**.** *Let S be a leftist circuit and $K \subseteq \mathcal{I}(S)$ be triangular with $|K| = k$ and $B \coloneqq B(K, S)$. Furthermore, let S' be the leftist circuit arising from S by deleting the input vertices, and let*

$$K' \coloneqq \left\{ v \in \mathcal{V}(S) \;\middle|\; \mathrm{depth}(v) = 1 \text{ and } w \in K \text{ for all } w \in \delta^-(v) \right\} \ .$$

Then the following statements are fulfilled.

(i) We have $|B| \leq k$.

(ii) We have $B = B(K', S') \cup (K \cap B)$.

(iii) The set K' is triangular with respect to S'.

(iv) For $k \geq 2$ we have $|K \cap B| \leq 2$, and $|K \cap B| \equiv k \pmod 2$, and

$$|K'| = \begin{cases} \frac{k-1}{2} & \text{if } k \text{ odd} \ ; \\ \frac{k}{2} & \text{if } k \text{ even, } |K \cap B| = 0 \ ; \\ \frac{k-2}{2} & \text{if } k \text{ even, } |K \cap B| = 2 \ . \end{cases}$$

Lemma 3.6.17. *Table 3.5 shows an upper bound on the* $\mathrm{AddSize}(n, m)$ *for every m and every n for which Algorithm 3.4 yields a circuit of depth at most $d \leq 4$.*

Proof. Table 3.5 shows the achieved additional size for (n, m) in green, and the split by which this is achieved in red. We distinguish different cases for m and the applied splits.

Case 1. $m = 0$

Proof. We start by proving the additional sizes for $m = 0$. For $n \leq 2$, we have $\mathrm{AddSize}(n, 0) = n - 1$ trivially. Let $n \geq 3$. We study the set $K = \{s_0, \ldots, s_{n-1}\}$. If n is odd, we compare the circuit for K to the one for $\frac{n-1}{2}$. Consider the set K' as defined in Lemma 3.6.16, implying that $|K'| \leq \frac{n-1}{2}$. Thus, the number of those boundary vertices for n that have depth at least 1 is precisely $|B(K', S')|$. Moreover,

m \ n	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
0	0,	0,	1,	1,	2,	2,	3,	2,	3,	3,	4,	3,	4,	4,	5,	3,	4,
1	0,	1,	1,	2, S2	2, S4	3, S4	2, S4	3, S4	3, S8	4, S8	3, S8	4, S8	4, S8	5, S8	3, S8	4, S8	
2	1,	2,	3, S2	3, S3	4, S4	5, S4	5, S4	4, S7	5, S8	6, S8	7, S8	7, S8	7, S8	8, S8	8, S8		
3	2,	3, A1	3, A1	4, S3	5, S4	6, S4	5, S4	5, S7	6, S8	7, S8	7, S8	8, S8	8, S8	9, S8			
4	3, A1	4, A1	4, A1	5, A1	5, A1	6, S5	6, S4	6, S7	7, S8	8, S8	8, S8	9, S8					
5	5, A2	6, A3	5, A1	7, S3	8, S4	8, S5	9, S6	8, S7	9, S8	10, S8							
6	6, A3	7, A3	8, S2	8, S3	9, S4	9, S5	10, S6	9, S7	10, S8	11, S8							
7	7, A3	8, A3	8, A1	9, A1	9, A1	10, A1	9, A1	10, A1									
8	9, A3	10, A3	10, A3	11, A3	12, A3	13, A3											
9	10, A3	11, A3	11, A3	12, A3	13, A3	14, A3											
10	12, A6																

Table 3.5: This table shows for each number m of alternating inputs and each number n of symmetric inputs the achieved additional size $\text{AddSize}(n, m)$. The colors indicate the depths of the computed circuits (depth 1 in red, depth 2 in orange, depth 3 in yellow, depth 4 in green). See Table 3.3 for more details on the depth bounds.

m \ n	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
0	-1	0	1	1	2	3	3	4	4	5	5	5	5	6	6	6	6
1	0	1	2	2	3	4	4	5	5	6	6	6	6	7	7	7	
2	1	2	3	3	4	5	5	6	6	7	7	7	7	8	8		
3	2	3	4	4	5	6	6	7	7	8	8	8	8	9			
4	4	5	6	6	7	8	8	9	9	10	10	10					
5	5	6	7	7	8	9	9	10	10	11							
6	6	7	8	8	9	10	10	11	11	12							
7	8	9	10	10	11	12	12	13									
8	9	10	11	11	12	13											
9	10	11	12	12	13	14											
10	12																

Table 3.6: This table shows for each number m of alternating inputs and each number n of symmetric inputs the value $\lfloor \psi(d, n, m) \rfloor$. The colors follow the scheme from Table 3.5. As one can check, we have $\text{AddSize}(n, m) \leq \lfloor \psi(d, n, m) \rfloor$ for all values (n, m) contained in Table 3.5.

we have $B(K', S') = B\Big(\{s_0, \ldots, s_{\frac{n-1}{2}-1}\}, S\Big) = \text{AddSize}(\frac{n-1}{2}, 0) + 1$. If n is odd, then there is exactly one boundary vertex of depth 0. Hence, it is

$$\text{AddSize}(n, 0) = B(K, S) - 1 = B\big(K', S'\big) + 1 - 1 = \text{AddSize}\left(\frac{n-1}{2}, 0\right) + 1 \ . \tag{3.22}$$

Similarly, if n is even, we have $|K'| = \frac{n}{2} - 1$ or $|K'| = \frac{n}{2} - 1$ by Lemma 3.6.16. The number of boundary vertices for n of depth at least 1 is precisely $B(K', S')$. As before, we have $B(K', S') = B(\{s_0, \ldots, s_{|K|}\}, S) = \text{AddSize}(|K|)+1$. If $|K'| = \frac{n}{2}-1$, we have two additional boundary vertices of depth 0 and this implies

$$\text{AddSize}(n, 0) = B(K, S) - 1 = B\big(K', S'\big) + 2 - 1 = \text{AddSize}\left(\frac{n-2}{2}, 0\right) + 2 \ . \tag{3.23}$$

Else, it is $|K'| = \frac{n}{2} - 1$ without additional boundary vertices and thus,

$$\begin{aligned} \text{AddSize}(n, 0) = B(K, S) - 1 &= B\big(K', S'\big) - 1 \\ &= \text{AddSize}\left(\frac{n}{2}, 0\right) \\ &\leq \text{AddSize}\left(\frac{n-2}{2}, 0\right) + 2 \ , \end{aligned}$$

since the AddSize can increase by at most one when increasing n by one. Checking the numbers for $m = 0$ in Table 3.5 yields that indeed the claimed sizes are just recursively computed using Equation (3.22) and Equation (3.23). □

Case 2. $m = 1$

Proof. Lemma 3.6.9 proves that indeed $s \dotplus t_0$ is triangular, and thus we have $\text{AddSize}(n, 1) = \text{AddSize}(n + 1, 0)$. These are precisely the values given for $m = 1$. □

Case 3. Very small cases ($m + n \leq 3$).

Proof. For $m+n \leq 3$ we either have $n \leq 1$ and then the build circuit is the standard And-Or path representation and $\text{AddSize}(n, m)$ is just $m+n-1$. For $(n, m) = (2, 1)$, we know by Lemma 3.6.9 that (s_0, s_1, t_0) is triangular and thus, we can build this subcircuit reusing one gate from the leftist circuit. Thus, $\text{AddSize}(2, 1) = 1$ indeed. □

Case 4. Alternating split for $m \geq 2$.

Proof. For every pair (n, m), for which we apply an alternating split, the additional size can be bounded by sum of the additional sizes of the subcircuits plus one for the concatenation gate as described in Observation 3.6.11. Indeed, all the numbers for which an alternating split is given are just calculated like this. □

Case 5. Symmetric split for $m \geq 2$.

Proof. For most of the symmetric split, the sizes are implied by Observation 3.6.11. Sometimes though, this size is not best possible. For these pairs (n, m) the size claimed in the table is implied by Lemma 3.6.15. To be more precise, these pairs are the symmetric splits for $n = 6$ at $k = 4$ for which $\rho(6) - \rho(2) = 2$, for $n = 11$

at $k = 8$ for which $\rho(11) - \rho(3) = 4$, and for $n \in \{12, 13, 14\}$ at $k = 8$ for which $\rho(n) - \rho(n-k) = 3$. □

Comparing to Table 3.5 shows that this indeed covers all the contained cases. □

Lemma 3.6.18. *Let $n, m \in \mathbb{Z}_{\geq 0}$ such that Algorithm 3.4 returns a circuit with depth $d \leq 4$. Then* $\mathrm{AddSize}(n, m) \leq \psi(d, n, m)$.

Proof. By Lemma 3.6.17 Table 3.5 shows an upper bound on the additional sizes. Table 3.6 shows the values of $\lfloor \psi(d, n, m) \rfloor$ for $d \leq 4$ for (n, m) for which Algorithm 3.4 computes a circuit of depth $d \leq 4$. A detailed comparison of the values of $\psi(d, n, m)$ and $\mathrm{AddSize}(n, m)$ indeed yields that $\mathrm{AddSize}(n, m) \leq \psi(d, n, m)$. □

This basically concludes the proof that $\mathrm{AddSize}(n, m) \leq \psi(d, n, m)$. Before formally putting everything together, we need to evaluate the size bound $\lim_{d\to\infty} \beta_d$ that we have just proven. For this, we start by a simple lemma.

Lemma 3.6.19. *For $d \in \mathbb{Z}_{\geq 2}$, let $S_d \coloneqq \sum_{i=2}^{d} \frac{i(i-1)}{2^{i-2}}$. Then, we have*

$$S_d = 16 - \frac{4}{2^d}\Big(d^2 + 3d + 4\Big) \ .$$

Proof. We apply induction in d. For $d = 2$, both terms are 2. For the induction step from d to $d + 1$ consider

$$\begin{aligned} S_{d+1} &= \sum_{i=2}^{d} \frac{i(i-1)}{2^{i-2}} \\ &\overset{\text{(IH)}}{=} 16 - \frac{4}{2^d}\Big(d^2 + 3d + 4\Big) + \frac{(d+1)d}{2^{d-1}} \\ &= 16 - \frac{4}{2^{d+1}}\Big(d^2 + 5d + 8\Big) \\ &= 16 - \frac{4}{2^{d+1}}\Big((d+1)^2 + 3(d+1) + 4\Big) \ . \end{aligned}$$ □

Lemma 3.6.20. *For $d \in \mathbb{Z}_{\geq 5}$, let $\beta_d \coloneqq 1.3 + \sum_{i=5}^{d} \frac{(i-1)}{\xi_{i-1} 2^{i-2}} \left(i + \log_2\left(\frac{\xi_{i-1}}{3(i-1)} \right) \right)$. Then, we have $\beta_d \leq 3.08$ for all $d \in \mathbb{Z}_{\geq 5}$.*

Proof. We compute upper bounds for β_d for small values of d explicitly in Table 3.7.

Lemma 3.6.19 implies that $\sum_{i=2}^{\infty} \frac{i(i-1)}{2^{i-2}} = 16$. Moreover, we get

$$\begin{aligned} \sum_{i=19}^{\infty} \frac{i(i-1)}{2^{i-2}} &= \sum_{i=2}^{\infty} \frac{i(i-1)}{2^{i-2}} - \sum_{i=2}^{18} \frac{i(i-1)}{2^{i-2}} \\ &= 16 - \left(16 - \frac{4}{2^{18}}\Big(18^2 + 3 \cdot 18 + 4\Big)\right) = \frac{4 \cdot 388}{2^{18}} \leq 0.006 \ . \end{aligned} \tag{3.24}$$

d	β_d
5	1.84740
6	2.25927
7	2.54997
8	2.74562
9	2.87258
10	2.95262
11	3.00193
12	3.03173
13	3.04946
14	3.05986
15	3.06590
16	3.06937
17	3.07135
18	3.07247

Table 3.7: Upper bounds for values of β_d for small values of d

Therefore, we have (using the values from the table above)

$$\begin{aligned}
\beta_d &= 1.3 + \sum_{i=5}^{d} \frac{(i-1)}{\xi_{i-1}2^{i-2}} \left(i + \log_2 \left(\frac{\xi_{i-1}}{3(i-1)} \right) \right) \\
&\leq 1.3 + \sum_{i=5}^{\infty} \frac{(i-1)}{\xi_{i-1}2^{i-2}} \left(i + \log_2 \left(\frac{\xi_{i-1}}{3(i-1)} \right) \right) \\
&\leq 3.07247 + \sum_{i=19}^{\infty} \frac{(i-1)}{\xi_{i-1}2^{i-2}} \left(i + \log_2 \left(\frac{\xi_{i-1}}{3(i-1)} \right) \right) \\
&\leq 3.07247 + \sum_{i=19}^{\infty} \frac{i(i-1)}{\xi_{i-1}2^{i-2}} \\
&\leq 3.07247 + \sum_{i=19}^{\infty} \frac{i(i-1)}{2 \cdot 2^{i-2}} \\
&\overset{(3.24)}{\leq} 3.07247 + 0.003 \\
&= 3.07647 \ .
\end{aligned}$$

□

Theorem 3.6.21. *Let $s = (s_0, \ldots, s_{n-1})$ and $t = (t_0, \ldots, t_{m-1})$ be input variables with $m \geq 1$. Algorithm 3.5 computes a circuit $C(s,t)$ for the extended* And-Or *path $f(s,t)$ with size*

$$\text{size}(C(s,t)) \leq 4.08m + n + \rho(n) - 3$$

where $\rho(n) = \begin{cases} n & \text{if } n \leq 2 \\ \lfloor 2\log_2(n-1) \rfloor & \text{if } n \geq 3 \ . \end{cases}$

Proof. We combine Lemma 3.6.14, Lemma 3.6.12, Lemma 3.6.13 and Lemma 3.6.20 to prove the statement by induction. Lemma 3.6.18 shows that $\text{AddSize}(n,m) \leq \psi(d,n,m)$ for $d \leq 4$ For larger $d \geq 5$, we either perform a symmetric split

with a part or all of the prefix, or we perform an alternating split. In each of the cases, we apply Lemma 3.6.12, Lemma 3.6.13 or Lemma 3.6.14 to see that $\text{AddSize}(n,m) \leq \psi(d,n,m)$. This proves $\text{AddSize}(n,m) \leq \psi(d,n,m)$ for all $d \in \mathbb{Z}_{\geq 1}$. By Lemma 3.6.20, we have $\beta_d \leq 3.08$ for all $d \in \mathbb{Z}_{\geq 1}$. The circuits computed by Algorithm 3.4 consist of the additional gates and the leftist circuits. Both leftist circuits S_0 and S_1 together form a forest with at least two connected components, implying that their total size is at most $n+m-2$. This gives a total size of at most

$$\begin{aligned}\text{size}(C(s,t)) \leq \psi(d,n,m) + m - 2 &\leq 3.08m + \rho(n) - 1 + m + n - 2 \\ &= 4.08m + n + \rho(n) - 3 \ . \quad \square\end{aligned}$$

Corollary 3.6.22. *Given input variables $t = (t_0, \ldots, t_{m-1})$ with $m \geq 2$, Algorithm 3.5 computes a circuit C for the AND-OR path $g(t)$ with depth*

$$\begin{aligned}\text{depth}(C(t)) \leq \log_2 m - \log_2\left(4 - \frac{3}{(\log_2 m + \log_2\log_2 m - 2)^{0.5}}\right) \\ + \log_2(\log_2 m + \log_2\log_2 m) + 1\end{aligned}$$

and size

$$\text{size}(C) \leq 4.08m - 3$$

in running time $\mathcal{O}(m \log_2 m)$.

Proof. Proposition 3.6.8 shows that the given depth bound is implied by Corollary 3.5.2. The bound on the size is implied by Theorem 3.6.21 for $n = 0$. It remains to analyze the running time. The construction of the leftist circuits takes only linear time. Since we build a single gate in every recursive call to the algorithm, there are at most $\mathcal{O}(m)$ iterations. Let n' and m' be the numbers of symmetric and alternating inputs within a recursive call. The depth computation in line 3 takes at most $\mathcal{O}(\log_2\log_2(m'+n'))$ time if it is implemented by a binary search. When applying a symmetric split, both the triangular set computation and the construction of the symmetric tree on the triangular subset can be applied in $\mathcal{O}(\log_2(n'))$ time, as already shown by Brenner and Silvanus [BS24]. The alternating split can be performed in constant time. Since $n' \leq m$ and $m' \leq m$ for every subinstance, this implies that the total running time is $\mathcal{O}(m \log_2 m)$. $\square$

3.6.4 Computer-aided size improvement

Our proof for the size analysis consists of an induction start for $d \leq 4$ and then an induction step which is in fact independent of β_0. One can improve the analysis by making a later induction start and then using the same induction steps as before. Thereby, we lose less on small circuits which helps significantly due to recursion. We use computational effort to derive an induction start for $d \leq 16$.

We modify Algorithm 3.5 as follows. If a circuit of depth at most 16 is computed, we try all possible splits leading to this depth, and choose the split giving the minimum AddSize. For those instances, we basically run Algorithm 3.1 and among all depth-optimum circuits, we choose the one achieving the minimum AddSize. For larger instances we apply the recursion until we reach a depth of at most 16 and then apply the above.

Improved bounds on the additional size in special cases

We need to be able to compute a good bound on the AddSize of our small circuits to improve on the previous size estimation. In general, the additional size of a split is bounded by one plus the sum of the additional sizes of the two subcircuits as already remarked in Observation 3.6.11. For alternating splits, we stick to precisely this bound. For symmetric splits, we can improve on this in various cases. In the following we give three improvements for the bounds used for symmetric splits.

Lemma 3.6.23. *We have*

$$\mathrm{AddSize}(n, 3) \leq \mathrm{AddSize}(0, n+1) + 2 \ .$$

Proof. Assume we are given Boolean variables $s = (s_0, \ldots, s_{n-1})$ and $t = (t_0, t_1, t_2)$ and a leftist circuit S. We prove by induction that the actual additional size to build (s, t) using S is bounded by $|B(s +\!\!+ (t_0), S)| + 1$. Here, we use $s +\!\!+ (t_0) = (s_0, \ldots, s_{n-1}, t_0)$ for the concatenation and $B(s +\!\!+ (t_0), S)$ for its set of boundary vertices with respect to S.

For the induction start, we can check that this is true for $n \leq 2$ using table Table 3.5. Indeed, for $n = 0$ we need an additional size of 2; and for $n = 1$ and $n = 2$ we apply an alternating split for $k = 1$ which corresponds to splitting off $s +\!\!+ (t_0)$. This directly allows us to combine the boundary vertices and derive the additional summand of 2 for adding the remaining two inputs.

Let $n \geq 3$. Then Algorithm 3.4 always applies a symmetric split splitting off a triangular subset $s' \subseteq s +\!\!+ (t_0)$ using Algorithm 3.3. We know by induction that we need at most $|B((s +\!\!+ (t_0)) \setminus s', S)| + 1$ additional vertices to build the circuit for $(s +\!\!+ (t_0)) \setminus s', (t_1, t_2)$. For the And circuit on s' we need $|B(s', S)| - 1$ gates, and then we need one additional gate to combine the two circuits. Since $B(s', S) \cup B((s +\!\!+ (t_0)) \setminus s', S)$ is a partition of $B(s +\!\!+ (t_0), S)$, this implies that the total additional size is at most $|B(s +\!\!+ (t_0), S)| + 1$ vertices as desired.

To conclude, we just observe, that $\mathrm{AddSize}(0, n+1) \geq |B(s +\!\!+ (t_0), S)| - 1$ for any s and S. Moreover, there exists some s, t for fixed S such that $\mathrm{AddSize}(n, 3) = |B(s +\!\!+ (t_0), S)| - 1$. Thus, this tuple s, t yields

$$\mathrm{AddSize}(n, 3) \leq |B(s +\!\!+ (t_0), S)| + 1 \leq \mathrm{AddSize}(0, n+1) + 2 \ .$$

□

Lemma 3.6.24. *Let $n, m \in \mathbb{Z}_{\geq 0}$ such that n is even and such that Algorithm 3.4 applies a symmetric split with the whole prefix in line 13. Then*

$$\mathrm{AddSize}(n, m) \leq \mathrm{AddSize}(0, m) + \mathrm{AddSize}(n+1, 0) \ .$$

Proof. We know that Algorithm 3.4 splits off n symmetric inputs. Due to Lemma 3.6.9, we know that we can choose these n inputs from the triangular set $s +\!\!+ t_0$, i.e., from a triangular set with $n+1$ inputs. We choose a triangular subset s' with the minimum number of boundary vertices as usual. The number of boundary vertices for these $n+1$ inputs is at most $\mathrm{AddSize}(n+1, 0)+1$. As $n+1$ is odd, there is exactly one boundary vertex which is an input itself. This is the boundary vertex that remains for the remaining instance. Thus, the number of boundary vertices for the symmetric prefix s' is precisely $\mathrm{AddSize}(n+1, 0)$ and the additional

size to build the circuit for it is $\mathrm{AddSize}(n+1)-1$. In total, this yields a total additional size of at most

$$\mathrm{AddSize}(0,m)+\mathrm{AddSize}(n+1)-1+1=\mathrm{AddSize}(0,m)+\mathrm{AddSize}(n+1)\ . \quad \square$$

Lemma 3.6.25. *Let* $\mathrm{lb}\colon \mathbb{Z}_{\geq 1}\to\mathbb{Z}_{\geq 0}$ *be a function such that* $\mathrm{lb}(n)$ *lower bound on the additional size needed to build a circuit for* n *symmetric inputs. Let* $n,m\in\mathbb{Z}_{\geq}0$ *such that Algorithm 3.4 applies a symmetric split with* $k<n$*. Then*

$$\mathrm{AddSize}(n,m)\leq \mathrm{AddSize}(n-k,m)+\mathrm{AddSize}(n,0)-\mathrm{lb}(n-k)\ .$$

Proof. Let (s,t) be the inputs at this step of the algorithm. The algorithm is working with triangular sets only and thus s is triangular. We split off a subset s' of s whose size is the largest power of two, i.e., k is a power of two such that $k<n<2k$. To determine this subset s', we use Algorithm 3.3. During this algorithm the boundary vertices of $B(s,S)$ get partitioned, i.e., $B(s',S)\cup B(s\setminus s',S)=B(s,S)$ because the algorithm determines s' as the inputs of a subset of the boundary tree sequence. Thus, we have in particular

$$|B(s',S)|=|B(s,S)|-|B(s\setminus s',S)|\ .$$

Moreover, we have $|B(s,S)|\leq \mathrm{AddSize}(n,0)+1$ and $|B(n\setminus s',S)|\geq \mathrm{lb}(n-k)$ due to the lower bound property. Using this, we can see that when applying the corresponding split, we get

$$\begin{aligned}\mathrm{AddSize}(n,m)&\leq |B(s',S)|-1+\mathrm{AddSize}(n-k,m)+1\\&=\mathrm{AddSize}(n-k,m)+|B(s,S)|-|B(s\setminus s',S)|\\&\leq \mathrm{AddSize}(n-k,m)+\mathrm{AddSize}(n,0)-\mathrm{lb}(n-k)\ .\end{aligned}\quad \square$$

For Lemma 3.6.25 we need a large lower bound. The bound that we use is given by the following lemma.

Lemma 3.6.26. *Let* s *be* n *consecutive input variables and* $b(n)$ *be the number of ones of* n *in binary representation. Then* $|B(s,S)|\geq b(n)$ *for any leftist circuit* S *on a superset of* s*.*

Proof. The boundary vertices $B(s,S)$ contain exactly one vertex of depth d if and only if n has a one in binary representation at position d. Else, they contain either two or no vertices of depth d. This implies that there are at least $b(n)$ boundary vertices. $\square$

Corollary 3.6.27. *For* $n\in\mathbb{Z}_{\geq 1}$ *let* $b(n)$ *the number of ones of* n *in binary representation. Let* $n,m\in\mathbb{Z}_{\geq}0$ *such that Algorithm 3.4 applies a symmetric split with* $k<n$*. Then*

$$\mathrm{AddSize}(n,m)\leq \mathrm{AddSize}(n-k,m)+\mathrm{AddSize}(n,0)-b(n-k)\ .$$

Whenever we consider a symmetric split, we get we derive the AddSize as the minimum of the bound implied by any Observation 3.6.11, Lemma 3.6.23, Lemma 3.6.24 and Corollary 3.6.27. Note that not for all splits every of these statements is applicable, we just consider those suitable for the current input and prefix numbers.

Size bound for an improved induction start

For values $d \leq 4$, one can check all of these values by hand as we have done in Table 3.5. A quick comparison against Table 3.6 showed that these are indeed correct values. For larger values of d, this can only be done automatically by a computer. Our implementation of the Algorithm 3.1 is always tracking an upper bound on the additional size for every (n, m), and recursively computing these along with the derived split for every pair (n, m) of input numbers. Additionally, it computes for every (n, m) the minimum possible value to β such that $\text{AddSize}(n, m) \leq \beta m + \rho(n) - 1$ still holds. In the end, it returns the maximum value for β found throughout the algorithm.

For $d \leq 16$ this value turns out to be $\beta_0 \leq 1.617$ which is attained at $m = 5723$ and $n = 12888$. The computation for this has been executed on a Rocky Linux 8 4.18 machine with an AMD EPYC 9684X processor. It took 33 hours and 57 minutes. The resulting β_0 implies that for every $d \leq 16$ we have

$$\text{AddSize}(n, m) \leq 1.617m + \rho(n) - 1 \ .$$

Using this as induction start, we can define a significantly smaller

$$\beta'_d \coloneqq 1.617 + \sum_{i=17}^{d} \frac{(i-1)}{\xi_{i-1} 2^{i-2}} \left(i + \log_2 \left(\frac{\xi_{i-1}}{3(i-1)} \right) \right) \ . \tag{3.25}$$

To evaluate the limit $\lim_{d \to \infty} \beta_d$, we recall that Equation (3.24) shows that

$$\sum_{i=19}^{\infty} \frac{i(i-1)}{2^{i-2}} \leq 0.006 \ .$$

From Table 3.7 we can further derive that in Equation (3.25) the summands for $i = 17$ and $i = 18$ are both at most 0.002. Since $\xi_d \geq 2$ for $d \geq 19$ this implies

$$\beta'_d \leq \beta_0 + 0.002 + 0.002 + 0.003 = 1.617 + 0.006 \leq 1.63 \ .$$

Thus, the true number of additional gates is bounded by

$$\text{AddSize}(n, m) \leq 1.63m + \rho(n) - 1 \ .$$

Hence, the constant 4.08 of Theorem 3.6.21 can be improved to 2.63 by a computer-aided proof.

Theorem 3.6.28. *Let $s = (s_0, \ldots, s_{n-1})$ and $t = (t_0, \ldots, t_{m-1})$ be input variables with $m \geq 1$. Algorithm 3.5 computes a circuit $C(s, t)$ for the extended* And-Or *path $f(s, t)$ with size*

$$\text{size}(C(s, t)) \leq 2.63m + n + \rho(n) - 3$$

where $\rho(n) = \begin{cases} n & \text{if } n \leq 2 \\ \lfloor 2 \log_2(n-1) \rfloor & \text{if } n \geq 3 \ . \end{cases}$

This implies the following corollary analogously to Corollary 3.6.22.

Corollary 3.6.29. *Given input variables $t = (t_0, \dots, t_{m-1})$ with $m \geq 2$, Algorithm 3.5 computes a circuit C for the AND-OR path $g(t)$ with depth*

$$\operatorname{depth}(C)\begin{cases} = \lceil \log_2 m \rceil & \text{for } m \leq 3 \text{ and} \\ \leq \log_2 m + \log_2 \log_2 m & \text{for } m \geq 4 \end{cases}$$

and size

$$\operatorname{size}(C) \leq 2.63m - 3$$

in running time $\mathcal{O}(m \log_2 m)$.

Note that this is equivalent to Theorem 3.1.3.

Corollary 3.6.30. *Given input variables $t = (t_0, \dots, t_{m-1})$ with $m \geq 2$, Algorithm 3.5 computes a circuit C for the AND-OR path $g(t)$ with depth*

$$\operatorname{depth}(C(t)) \leq \log_2 m - \log_2\left(4 - \frac{3}{(\log_2 m + \log_2 \log_2 m - 2)^{0.5}}\right) + \log_2(\log_2 m + \log_2 \log_2 m) + 1$$

and size

$$\operatorname{size}(C) \leq 2.63m - 3$$

in running time $\mathcal{O}(m \log_2 m)$.

As remarked in Remark 3.5.3, this depth tends to $\log_2 m + \log_2 \log_2 m - 1$ asymptotically and therefore Corollary 3.6.30 implies Theorem 3.1.4.

Note also that without introducing better bounds on the additional size in further cases, there is not much to gain with more computational effort. The general bound β' only increased by less than 0.02 compared to the bounded β_{16} needed for the induction start. This implies that the possible gain implied by more computational effort is also bounded by 0.02.

3.7 Faster adder circuits

In this section we use the AND-OR path circuits we derived to create fast adder circuits. As an adder needs to compute n carry bits, it can be naively constructed using n independent AND-OR paths, yielding a quadratic size. A quadratic size is not useful in practice though, so we apply recursion strategies similar to Hermann [Her20] and Brenner and Silvanus [BS24], in order to derive improved adders with a slightly worse depth bound and linear size though. As a first step, they showed how to use their AND-OR paths to yield near-linear size adders. Those are then later needed as subcircuits for linear size adders. We adapt their strategies since we also want to use such adders as subcircuits.

3.7.1 Near-linear size adders

The first step is a recursions step to build an adder from two smaller adders along with some side logic. To build the adder on n input pairs, one decomposes the input pairs into two halves. The adder on the right half correctly computes half of the carry bits. The other carry bits are obtained by alternating splits splitting off all but the right $\frac{n}{2}$ input pairs. The right part of all these splits are then computed by an AND-OR path circuit on the right inputs and an AND-prefix circuit on the left inputs. The left part of the split is computed by an adder circuit on the left inputs.

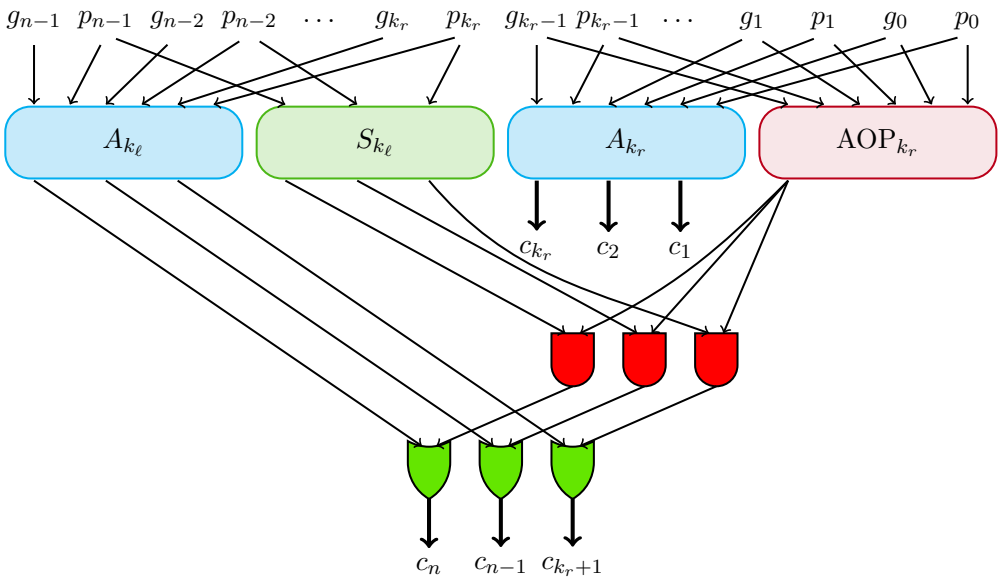


Figure 3.11: Illustration for Algorithm 3.6.

An illustration of this step can be seen in Figure 3.11. The described algorithm by Hermann is summarized in Algorithm 3.6. They prove the following lemma regarding its guarantee.

Lemma 3.7.1 (Hermann [Her20])**.** *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 $(\mathrm{AOP}_k)_{k<n}$, Algorithm 3.6 computes an adder circuit C_n with*

$$\mathrm{depth}(C_n) \leq \max\{\mathrm{depth}(A_{k_r}), \mathrm{depth}(A_{k_\ell}) + 1, \mathrm{depth}(\mathrm{AOP}_{k_r}) + 2, \mathrm{depth}(S_{k_\ell}) + 2\}$$

and

$$\mathrm{size}(C_n) \leq \mathrm{size}(A_{k_r}) + \mathrm{size}(A_{k_\ell}) + \mathrm{size}(\mathrm{AOP}_{k_r}) + \mathrm{size}(S_{k_\ell}) + 2k_\ell \ .$$

Using this recursion step, we can similarly to their analysis obtain a bound on our adders. Hermann [Her20] and Brenner and Silvanus [BS24] have used existing adder circuits in small cases that are no longer sufficient for our depth bound. Thus, we use for small n some adder circuits that can be obtained from our And-Or paths. To save some gates, we do not use disjoint And-Or paths for the different carry bits but allow ourselves to reuse the leftist circuit constructed for the last carry bit for all other And-Or paths. For large n, we apply Algorithm 3.6 recursively. This yields the following result.

Theorem 3.7.2. *Let $n \in \mathbb{N}$ with $n \geq 3$ and $c := 5.6$. We can construct an adder circuit A_n^1 on n input pairs in running time $\mathcal{O}\left(n \log_2^2 n\right)$ with depth*

$$\mathrm{depth}(A_n^1) \leq \log_2 n + \log_2 \log_2 n + 2$$

and size

$$\mathrm{size}(A_n^1) \leq cn \log_2 n \ .$$

Algorithm 3.6: 2-part adder construction framework by Hermann [Her20]

Input: $n \in \mathbb{N}$, $n \geq 2$, and n input pairs $p_0, g_0, \ldots, p_{n-1}, g_{n-1}$, 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 $p_0, g_0, \ldots, p_{n-1}, g_{n-1}$.

$k_\ell \leftarrow \lfloor \frac{n}{2} \rfloor$, $k_r \leftarrow \lceil \frac{n}{2} \rceil$.
$P_r \leftarrow (p_0, g_0, \ldots, p_{k_r-1}, g_{k_r-1})$, $P_\ell \leftarrow (p_{k_r}, g_{k_r}, \ldots, p_{n-1}, g_{n-1})$.
Compute an adder circuit A_{k_ℓ} on P_ℓ.
Compute an adder circuit A_{k_r} on P_r.
Compute an AND-prefix circuit S_{k_ℓ} on the inputs p_i with $i > k_r$ of P_{k_ℓ}.
Compute an AND-OR path circuit AOP_{k_r} on the inputs of P_{k_r}.
for $i \leftarrow 1$ **to** k_r **do**
  Let $\mathrm{out}_i(C_n) \coloneqq \mathrm{out}_i(A_{k_r})$.
for $i \leftarrow 1$ **to** k_ℓ **do**
  Let $\mathrm{out}_{k_r+i}(C_n) \coloneqq \mathrm{out}_i(A_{k_\ell}) \vee \left(\mathrm{out}_i(S_{k_\ell}) \wedge AOP_{k_r}\right)$.
return C_n.

Proof. We prove the depth and size bounds by induction on n.

Case 1. $n = 3$

Proof. For this case we use the ripple carry adder with a depth of $2n - 2 = 4 \leq \log_2(3) + \log_2 \log_2 3 + 2$ and size $2n - 2 = 4$ which clearly suffices. □

Case 2. Let $4 \leq n \leq 15$.

Proof. In these cases we use an adder that is just a collection of AND-OR paths. Instead of building them completely disjoint, we reuse their leftist circuit though. Thus, the total size of our adder is the size of the leftist circuit plus the sum over all additional sizes needed for our AND-OR paths.

To build an adder with n input pairs, we need $2n$ inputs. Thus, Corollary 3.6.29 shows that the depth of these AND-OR paths is at most

$$\log_2(2n) + \log_2 \log_2(2n) \leq \log_2(n) + \log_2 \log_2(n) + 2$$

for $n \leq 15$ as desired.

The leftist circuit is of size $2n - 2$ as analyzed in Theorem 3.6.21. To determine the additional sizes, we use the techniques introduced in Section 3.6.4. We show the additional sizes along with some other data in Table 3.8.

For a depth of at most 4, these sizes already implied by Table 3.5, and we just extend it to derive the additional sizes for up to 30 inputs. The initial column of Table 3.8 shows the number $2n$ of inputs we are studying. The first data column contains the additional size needed to build the number of inputs $2n$. The second data column is just a cumulative version of the first, adding up all the additional sizes for an adder with n inputs. The third column shows the total size for an adder with n inputs and the last column shows the bound $5.6n \log_2 n$. As desired, we see that building all these AND-OR paths does not lead to a larger size than desired for any $4 \leq n \leq 15$. □

Case 3. Let $n \geq 16$.

$2n$	AddSize	cum.	Adder	$\lfloor 5.6n \log_2 n \rfloor$
2	1	1	1	0
4	3	4	6	11
6	6	10	14	26
8	8	18	24	44
10	12	30	38	65
12	14	44	54	86
14	17	61	73	110
16	20	81	95	134
18	23	104	120	159
20	25	129	147	186
22	27	156	176	213
24	30	186	208	240
26	33	219	243	269
28	37	256	282	298
30	41	297	325	328

Table 3.8: Overview of the sizes constructed for Case 2 in the proof of Theorem 3.7.2. The initial column shows the input numbers $2n$ for which we construct an And-Or path. The first column then shows the additional sizes that are obtained for such And-Or paths, i.e., $\mathrm{AddSize}(0, 2n)$. The second column shows the cumulative additional size for all And-Or paths of at most $2n$ inputs, i.e., $\sum_{i=1}^{n} \mathrm{AddSize}(0, 2i)$. The third column displays the total size of an adder constructed in that fashion, that is the cumulative additional size added to the size $2n - 2$ for the common leftist circuit. The last column then computes the bound on the depth implied by the theorem.

Proof. We prove this by induction using Algorithm 3.6 and assume that the result is true for all $i < n$. To apply this algorithm, we need to provide some circuit classes. The adder circuits $(A_k)_{k<n}$ are provided inductively since we can build $A_{k_\ell} \coloneqq A^1_{k_\ell}$ and $A_{k_r} \coloneqq A^1_{k_r}$ with the stated properties. This yields depth and size bounds of

$$\begin{aligned} \mathrm{depth}(A_{k_\ell}) &\overset{k_\ell \le \frac{n}{2}}{\le} \log_2 n + \log_2 \log_2 n + 1 \\ \mathrm{depth}(A_{k_r}) &\overset{k_r \le n}{\le} \log_2 n + \log_2 \log_2 n + 2 \\ \mathrm{size}(A_{k_\ell}) &\le c k_\ell \log_2(k_\ell) \\ \mathrm{size}(A_{k_r}) &\le c k_r \log_2(k_r) \ . \end{aligned}$$

Moreover, we use the Ladner-Fischer And-prefix circuit for $f = 2$ derived in [LF80] to build S_{k_ℓ}. This is a valid choice as $f \le \lceil \log_2(k_\ell) \rceil$ for $n \ge 6$. Thus, this circuit has the properties

$$\begin{aligned} \mathrm{depth}(S_{k_\ell}) \le \lceil \log_2(k_\ell) \rceil \le \log_2 n + 2 &\overset{n \ge 16}{\le} \log_2 n + \log_2 \log_2 n \\ \mathrm{size}(S_{k_\ell}) \le 2.5 k_\ell \ . \end{aligned}$$

Our AND-OR path circuits come with

$$\begin{aligned}\operatorname{depth}(\mathrm{AOP}_{k_r}) &\leq \log_2 n + \log_2 \log_2 n \\ \operatorname{size}(\mathrm{AOP}_{k_r}) &\leq 2.7n - 3 \ .\end{aligned}$$

This yields a total depth of

$$\begin{aligned}\operatorname{depth}(A^1_n) &\leq \max\{d(A_{k_r}), d(A_{k_\ell}) + 1, \operatorname{depth}(\mathrm{AOP}_{k_r}) + 2, \operatorname{depth}(S_{k_\ell}) + 2\} \\ &\leq \log_2 n + \log_2 \log_2 n + 2 \ .\end{aligned}$$

To analyze the size, we use that since $n \geq 16$

$$\log_2(k_r) \leq \log_2\left(\frac{n+1}{2}\right) \leq \log_2 n - 1 + (\log_2(17) - \log_2(16)) \leq \log_2 n - 0.912 \ .$$

This implies

$$\begin{aligned}\operatorname{size}(A_{k_r}) + \operatorname{size}(A_{k_\ell}) &= ck_r \log_2(k_r) + ck_\ell \log_2(k_\ell) \\ &\leq c(k_r(\log_2 n - 0.912) + k_\ell(\log_2 n - 1)) \\ &\leq cn(\log_2 n - 0.912) \ .\end{aligned}$$

Putting the size together, this yields

$$\begin{aligned}\operatorname{size}(A^1_n) &\leq \operatorname{size}(A_{k_r}) + \operatorname{size}(A_{k_\ell}) + \operatorname{size}(\mathrm{AOP}_{k_r}) + \operatorname{size}(S_{k_\ell}) + 2k_\ell \\ &\leq cn(\log_2 n - 0.912) + 2.7n + 2.5k_\ell + 2k_\ell \\ &\overset{k_\ell \leq \frac{n}{2}}{\leq} cn \log_2 n - 0.912cn + 4.95n \\ &\leq cn\left(\log_2 n - 0.912 + \frac{4.95}{c}\right) \\ &\leq cn(\log_2 n - 0.912 + 0.885) \\ &< cn \log_2 n \ .\end{aligned}$$

Thus, this case follows by induction. □

The runtime analysis can be done similarly to Brenner and Silvanus. This concludes the proof. □

3.7.2 Linearizing the size

We use the linearization framework by Brenner and Silvanus [BS24]. It is based on Algorithm 3.7. It needs two families of adder circuits for which we take the circuits constructed in Theorem 3.7.2 and a family of linear depth adder circuits that are only used for small instances. Moreover, it needs a family of AND-prefix circuits for which we take Ladner-Fischer circuits [LF80] and a family of AND-OR path circuits for which we take the circuits constructed in Corollary 3.6.29 in Section 3.6.

The algorithm itself is in some way a generalization of Algorithm 3.6. It subdivides the input pairs into ℓ consecutive sets with about k inputs each. We choose $k = \lceil \log_2 n \rceil$ and $\ell = \lceil \frac{n}{k} \rceil$. For each such set of input pairs, we construct an adder and an AND-prefix circuit on this subset and construct an AND-OR path on the inputs right of it. Then these circuits get combined to the complete adder. One can find further details on it in [BS24] or [Her20].

Using this algorithm yields the following theorem.

Algorithm 3.7: l-part adder construction framework by [Her20]

Input: $n \in \mathbb{N}$, $n \geq 2$, n input pairs $p_0, g_0, \ldots, p_{n-1}, g_{n-1}$, 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 $p_0, g_0, \ldots, p_{n-1}, g_{n-1}$.

Choose $k \in \mathbb{N}_{>0}$ and $l \leftarrow \lceil n/k \rceil$.
for $j \leftarrow 0$ **to** $l-2$ **do**
  $P^{(j)} \coloneqq (p_{jk}, g_{jk}, \ldots, p_{(j+1)k-1}, g_{(j+1)k-1})$.
$P^{(l-1)} \coloneqq (p_{(l-1)k}, g_{(l-1)k}, \ldots, p_{n-1}, g_{n-1})$.
for $j \leftarrow 0$ **to** $l-1$ **do**
  $n_j \leftarrow |P^{(j)}|$, $N_j \leftarrow n_0 + \ldots + n_{j-1}$.
Construct an adder circuit $A^{(0)}_{n_0}$ on $P^{(0)}$.
for $i \leftarrow 1$ **to** n_0 **do**
  Let $\mathrm{out}_i(C_n) \coloneqq \mathrm{out}_i\left(A^{(0)}_{n_0}\right)$.
for $j \leftarrow 1$ **to** $l-1$ **do**
  Construct an adder circuit $A^{(j)}_{n_j}$ on $P^{(j)}$.
  Construct an AND-prefix circuit $S^{(j)}_{n_j}$ on $P^{(j)}$.
  Construct an AND-OR path circuit $AOP^{(j)}_{N_j}$ on the N_j input pairs in $P^{(j-1)}, \ldots, P^{(0)}$.
  Let $\mathrm{out}_{N_j}(C_n) \coloneqq AOP^{(j)}$.
  for $i \leftarrow 1$ **to** n_j **do**
    Let $\mathrm{out}_{N_j+i}(C_n) \coloneqq \mathrm{out}_i\left(A^{(j)}_{n_j}\right) \vee \left(\mathrm{out}_i\left(S^{(j)}_{n_j}\right) \wedge AOP^{(j)}_{N_j}\right)$.
return C_n.

Theorem 3.7.3 ([BS24], Theorem 4.7)**.** *Let $n \in \mathbb{Z}_{\geq 2}$, two families of adder circuits $(A_k)_{k\in\mathbb{N}}$ and $(B_\ell)_{\ell\in\mathbb{N}}$, a family of AND-prefix circuits $(S_k)_{k\in\mathbb{N}}$ and a family of AND-OR path circuits $(\mathrm{AOP}_k)_{k\in\mathbb{N}}$ be given. One can compute an adder circuit C_n on n inputs with depth*

$$\begin{aligned}\mathrm{depth}(C_n) \leq \max\big\{&\mathrm{depth}(A_k)+1,\\ &\mathrm{depth}(B_\ell)+\max\{\mathrm{depth}(\mathrm{AOP}_k),\mathrm{depth}(S_k)\}+2\big\}\end{aligned}$$

and size

$$\mathrm{size}(C_n) \leq \mathrm{size}(A_{n_0}) + \sum_{j=0}^{\ell-1}\Big(\mathrm{size}(A_{n_j}) + \mathrm{size}(AOP_{n_j}) + \mathrm{size}(B_\ell + 2n)\Big)\ .$$

Moreover, we use the following linear depth adders.

Proposition 3.7.4 ([BS24], Proposition 4.8)**.** *For each $n \in \mathbb{N}$ there is an adder circuit A_n with $\mathrm{size}(A_n) \leq 3.5n$ and $\mathrm{depth}(A_n) \leq n+2$.*

We need a couple of computational lemmas.

Lemma 3.7.5. *Given $n \geq 256$, $k = \lceil \log_2 n \rceil$ and $\ell = \lceil \frac{n}{k} \rceil$, we have*

$$\log_2 \ell + \log_2 \log_2 \ell + \log_2 k + \log_2 \log_2(2k) \leq \log_2 n + \log_2 \log_2 n + \log_2 \log_2 \log_2 n + 0.531 \ .$$

Proof. We bound the different terms. For $n \geq 256$ it is

$$\frac{n}{k} \geq \frac{n}{\log_2(n) + 1} \geq 28 \tag{3.26}$$

and thus

$$\log_2 \ell \leq \log_2\left(\frac{n}{k} + 1\right) \leq \log_2\left(\frac{n}{k}\right) + 0.051 = \log_2 n - \log_2 k + 0.051 \ .$$

Additionally, we have

$$\begin{aligned}
\log_2 \log_2(2k) &\leq \log_2(1 + \log_2(\log_2 n + 1)) \\
&\leq \log_2\left(1 + \log_2\left(\frac{9}{8} \log_2 n\right)\right) \\
&< \log_2(\log_2 \log_2 n + 1.17) \\
&< \log_2(1.39 \log_2 \log_2 n) \\
&< \log_2 \log_2 \log_2 n + 0.48 \ .
\end{aligned}$$

Putting everything together yields

$$\begin{aligned}
&\log_2 \ell + \log_2 \log_2 \ell + \log_2 k + \log_2 \log_2(2k) \\
\leq &\log_2 n + 0.051 + \log_2 \log_2 \ell + \log_2 \log_2(2k) \\
\leq &\log_2 n + 0.051 + \log_2 \log_2 n + \log_2 \log_2(2k) \\
< &\log_2 n + \log_2 \log_2 n + \log_2 \log_2 \log_2 n + 0.531 \ .
\end{aligned}$$

□

Lemma 3.7.6. *For $n \leq 256$ it is*

$$2\lceil \log_2 n \rceil \leq \log_2 n + \log_2 \log_2 n + \log_2 \log_2 \log_2 n + 5.5 \ .$$

Proof. We define the function

$$\nu \colon x \mapsto \log_2 \log_2 x + \log_2 \log_2 \log_2 x + 3.5 - \log_2 x$$

for $x \geq 4$. It suffices to show that

$$\nu(x) \geq 0 \ .$$

Note that for $x \leq 11 < 2^{3.5}$ and thereby $\log_2 x < 3.5$, this is clearly fulfilled. Thus, assume that $x \geq 12$. We compute the derivative of $\nu(x)$

$$\begin{aligned}
\frac{d}{dx}\nu(x) &= \frac{1}{\ln^2(2) x \log_2 x} + \frac{1}{\ln^3(2) x \log_2 x \log_2 \log_2 x} - \frac{1}{\ln(2) x} \\
&= \frac{\ln(2) \log_2 \log_2 x + 1 - \ln^2(2) \log_2 x \log_2 \log_2 x}{\ln^3(2) x \log_2 x \log_2 \log_2 x} \ .
\end{aligned}$$

It is negative because its denominator is always positive and its nominator is

$$\begin{aligned} & \ln(2)\log_2\log_2 x + 1 - \ln^2(2)\log_2 x \log_2\log_2 x \\ = \;& 1 + \ln(2)\log_2\log_2 x(1 - \ln(2)\log_2 x) \\ \overset{x\geq 12}{\leq} \;& 1 - \ln(2)\log_2\log_2 x \\ \overset{x\geq 12}{<} \;& 0 \ . \end{aligned}$$

Thus, for $10 \leq x \leq 256$, we have $\nu(x) \geq \nu(256) = 3+\log_2 3+3.5-8 \geq 0.08 > 0$. □

Lemma 3.7.7. *Given $n \geq 256$, $k = \lceil \log_2 n \rceil$ and $\ell = \lceil \frac{n}{k} \rceil$, we have*

$$(5.6\log_2 \ell - 2)\ell \leq 5.6n \ .$$

Proof. We have

$$\ell = \left\lceil \frac{n}{k} \right\rceil \overset{k\geq \log_2 n}{\leq} \left\lceil \frac{n}{\log_2 n} \right\rceil \leq \frac{n}{\log_2 n} + 1 \tag{3.27}$$

and thus

$$\begin{aligned} \log_2 \ell \quad & \overset{(3.27)}{\leq} \quad \log_2\left(\frac{n}{\log_2 n} + 1\right) \\ & \overset{n\geq 256}{\leq} \quad \log_2\left(1.032\frac{n}{\log_2 n}\right) \\ & \leq \quad \log_2 n - \log_2\log_2 n + 0.05 \ . \end{aligned} \tag{3.28}$$

Hence, we obtain

$$\begin{aligned} (5.6\log_2 \ell - 2)\ell \quad & \overset{(3.28)}{\leq} \quad \big(5.6(\log_2 n - \log_2\log_2 n + 0.05) - 2\big)\ell \\ & \overset{(3.27)}{\leq} \quad \big(5.6(\log_2 n - \log_2\log_2 n + 0.05) - 2\big)\left(\frac{n}{\log_2 n} + 1\right) \\ & = \quad 5.6n - \frac{5.6n}{\log_2 n}\log_2\log_2 n + \frac{0.28n}{\log_2 n} \\ & \qquad + 5.6\log_2 n - 5.6\log_2\log_2 n + 0.28 - \frac{2n}{\log_2 n} - 2 \\ & < \quad 5.6n - \frac{5.6n}{\log_2 n}\log_2\log_2 n + 5.6\log_2 n \\ & = \quad 5.6n + \frac{5.6n\log_2\log_2 n}{\log_2 n}\left(-1 + \frac{\log_2^2 n}{n\log_2\log_2 n}\right) \ . \end{aligned}$$

For $n \geq 16$ it is $\log_2 n \leq \sqrt{n}$ and thus $\frac{\log_2^2 n}{n\log_2\log_2 n} < 1$. So the bracket is nonnegative and the last term can be bounded from above by $5.6n$. □

This suffices to prove Theorem 3.1.5, which we here repeat for the sake of completeness.

Theorem 3.1.5. *Let $n \in \mathbb{Z}_{\geq 4}$. There exists an algorithm to construct an adder circuit A_n^2 on n input pairs in running time $\mathcal{O}(n\log_2 n)$ with depth*

$$\operatorname{depth}(A_n^2) \leq \log_2 n + \log_2\log_2 n + \log_2\log_2\log_2 n + 5.6$$

and size

$$\operatorname{size}(A_n^2) \leq 19n \ .$$

Proof. For $1 \leq n \leq 256$ we use the adder L_n^f by Ladner and Fischer [LF80] with $f = 0$. It yields a circuit with depth at most $2\lceil \log_2 n \rceil$ which suffices as shown in Lemma 3.7.6. Its size is bounded by $12n$ which suffices, too.

For $n \geq 257$ we use the linearization of Theorem 3.7.3 with the following circuits and $k = \lceil \log_2 n \rceil$ and $\ell = \lceil \frac{n}{k} \rceil$. We choose $(B_\ell)_{\ell \in \mathbb{N}} = (A_\ell^1)_{\ell \in \mathbb{N}}$ as derived in Theorem 3.7.2. For the adder family $(A_k)_k \in \mathbb{N}$ we take the circuit given by Proposition 3.7.4. The AND-prefix circuit family $(S_k)_{k \in \mathbb{N}}$ will be the Ladner-Fischer circuits [LF80] $(S_k^2)_{k \in \mathbb{N}}$ for $f = 2$, which works since $\lceil \log_2 k \rceil > 2$. For the AND-OR path circuits, we use the circuits given in Theorem 3.1.3. Putting this together yields the following properties.

$$\begin{array}{lcllcl} \operatorname{depth}(A_k) & \leq & k + 2 & \operatorname{size}(A_k) & \leq & 3.5k \\ \operatorname{depth}(B_\ell) & \leq & \log_2 \ell + \log_2 \log_2 \ell + 2 & \operatorname{size}(B_\ell) & \leq & 5.6\ell \log_2 \ell \\ \operatorname{depth}(\mathrm{AOP}_k) & \leq & \log_2(2k) + \log_2 \log_2(2k) & \operatorname{size}(\mathrm{AOP}_k) & \leq & 5.4k - 3 \\ \operatorname{depth}(S_k) & \leq & \lceil \log_2 k \rceil + 2 & \operatorname{size}(S_k) & \leq & 2.5k \end{array} \tag{3.29}$$

Before applying Theorem 3.7.3, we show the following inequalities to determine how to evaluate the contained maxima. Since $k \geq 8$ (following from $n \geq 257$) we have

$$\operatorname{depth}(S_k) \leq \lceil \log_2 k \rceil + 2 < \log_2 k + 1 + \log_2 \log_2(2k) = \log_2(2k) + \log_2 \log_2(2k) \ . \tag{3.30}$$

Moreover,

$$\begin{aligned} \operatorname{depth}(A_k) + 1 \leq k + 3 \leq \log_2 n + 4 < \\ \log_2 \ell + \log_2 \log_2 \ell + \log_2(2k) + \log_2 \log_2(2k) + 4 \ . \end{aligned} \tag{3.31}$$

Now Theorem 3.7.3 gives the following bound on the depth.

$$\begin{aligned} \operatorname{depth}(A_n^2) &\leq \max\big\{\operatorname{depth}(A_k) + 1, \\ &\qquad \operatorname{depth}(B_\ell) + \max\{\operatorname{depth}(\mathrm{AOP}_k), \operatorname{depth}(S_k)\} + 2\big\} \\ &\leq \log_2 \ell + \log_2 \log_2 \ell \log_2(2k) + \log_2 \log_2(2k) + 4 \\ &\leq \log_2 \ell + \log_2 \log_2 \ell + \log_2 k + \log_2 \log_2(2k) + 5 \\ &\overset{\text{Lem. 3.7.5}}{\leq} \log_2 n + \log_2 \log_2 n + \log_2 \log_2 \log_2 n + 5.531 \ . \end{aligned}$$

The total size can be bounded by

$$\begin{aligned} \operatorname{size}(A_n^2) &\overset{\text{Thm. 3.7.3}}{<} \sum_{j=0}^{\ell-1} \Big(\operatorname{size}(A_{n_j}) + \operatorname{size}(S_{n_j}) + \operatorname{size}(\mathrm{AOP_{n_j}}) \Big) + \operatorname{size}(B_\ell) + 2n \\ &\overset{(3.29)}{\leq} \sum_{j=0}^{\ell-1} \big(3.5n_j + 2.5n_j + 5.4n_j - 2 \big) + 5.6\ell \log_2 \ell + 2n \\ &\leq 11.4n - 2\ell + 5.6\ell \log_2 \ell + 2n \\ &= 13.4n + (5.6\ell \log_2 \ell - 2)\ell \\ &\overset{\text{Lem. 3.7.7}}{\leq} 13.4n + 5.6n \\ &= 19n \ . \end{aligned}$$

Regarding the running time, we only need to consider $n \geq 257$. The adder circuit B_l on $\ell = \lceil \frac{n}{k} \rceil$ inputs can be constructed in time $\mathcal{O}(\ell \log_2 \ell) = \mathcal{O}(n \log_2 n)$

using Theorem 3.7.2. For the construction of the ℓ And-Or paths AOP_k, we use the algorithm of Corollary 3.6.29 with a total running time of $\mathcal{O}(\ell k \log_2 k) = \mathcal{O}(n \log_2 \log_2 n)$. The remaining part of the algorithm including the computation of the And-prefix circuit family S_k and the adder family A_k can be done in linear time. Thus, we get an overall running time of $\mathcal{O}(n \log_2 n)$. □

3.8 Our algorithm in practice

Our algorithm for restructuring And-Or paths can be implemented in a recursive manner. We want to show the depths achieved by Algorithm 3.1. Since this algorithm always performs the optimum split, the achieved depth is at most the one we have proven above. Trying all splits comes at the cost of a slightly increased running time. However, Table 3.9 shows the running times still being very fast even for instances of extremely large size. The data for this was collected by executing single-threaded runs on a Rocky Linux 8 4.18 machine with an AMD EPYC 9684X processor.

#inputs	Runtime [s]	Depth	Global bound	Asym. bound
64	< 0.01	8	8	7
128	< 0.01	9	9	8
256	< 0.01	10	11	10
512	0.02	11	12	11
1024	0.09	12	13	12
2048	0.35	13	14	13
4096	1.42	14	15	14
8192	5.89	15	16	15
16384	24.23	16	17	16
32768	97.82	18	18	17
65536	390.49	19	20	19
131072	1575.68	20	21	20
262144	6914.76	21	22	21

Table 3.9: This table shows the running times (without size optimization) and the optimum depths that can be achieved by splitting along the best split. Moreover it shows the global and the asymptotic bound on the depth that we have proven in this chapter.

Moreover, Table 3.9 shows that the achieved depth is indeed for larger m the depth we showed to be the asymptotic upper bound. Therefore, our analysis cannot easily be improved, at least not for instances with less than $200,000$ inputs.

3.9 Conclusions

We presented new And-Or path circuits with smaller depths that reduce the difference to the known lower bounds significantly. In particular, for large instances, the remaining gap to the lower bounds is arbitrarily close to 2. We have showed how to linearize the size of the And-Or paths. Moreover, we have used our And-Or path circuits to derive faster adder circuits with a linear size. Nevertheless, it remains an open question what the exact value of the optimum depth of And-Or path circuits is. This should be subject to further research since faster And-Or path circuits will probably lead to faster linear size adders, too.

Chapter 4

Multi-output And function minimization

In the previous chapter we had many single-output AND functions and constructed a leftist circuit to derive linear size AND-OR paths and adders. In this way we were able to reduce the total size of these functions significantly. We want to tie in with this and study multi-output AND functions in this chapter in general. Our previous AND functions had a very special structure since they were derived from consecutive sets. On real world chips we do not necessarily have this special structure for all instances. We turn to general subsets of a given base set and explore the problem of constructing a small circuit for a multidimensional AND function, i.e., a function $f\colon \{0,1\}^n \to \{0,1\}^m$ (for $n, m \in \mathbb{Z}_{>0}$) in which each component f_j is given by a single AND function on some of the input variables, i.e., $f_j = \bigwedge_{i \in T_j} x_i$ where $T_j \subseteq \{1, \ldots, n\}$. For such a function we aim to find a circuit computing f with the minimum number of AND operations.

We start by giving a formal definition of the problem along with some more motivation and background.

Parts of this chapter have previously been published in Armbruster [Arm24]. More details on the published parts can be found at the end of the introduction (see page 82).

4.1 Introduction to And function minimization

Our main problem formulation is as follows[1].

k-And Tree Size Optimization (k-ATSO):	
Instance:	a set of variables $(x_1, \ldots, x_n)$ and a set of index subsets $\mathcal{T} \subseteq \mathcal{P}(\{1, \ldots, n\})$ satisfying $\lvert T \rvert \leq k$ for all $T \in \mathcal{T}$.
Task:	find a circuit over AND2 computing $\bigwedge_{i \in T} x_i$ for each $T \in \mathcal{T}$ with the minimum number of used gates.

The parameter k bounds the maximum number of inputs of a single AND function. This is motivated by practical application: the depth of a circuit on chips cannot be too large because it is directly related to the speed of the chip. Thus, the size of any $T \in \mathcal{T}$ is also bounded. In general, depth is also an important objective for

[1] Notation: We use $\mathcal{P}(X)$ for the power set of a base set X.

logic optimization on chips. It is not a part of the objective in our main problem formulation, but we derive a partial result about the following problem nevertheless.

DEPTH OPTIMUM k-AND TREE SIZE OPTIMIZATION (k-DATSO):

Instance: a set of variables $(x_1, \ldots, x_n)$ and a set of index subsets $\mathcal{T} \subseteq \mathcal{P}(\{1, \ldots, n\})$ satisfying $|T| \leq k$ for all $T \in \mathcal{T}$.

Task: find a circuit over AND2 computing $\bigwedge_{i \in T} x_i$ for each $T \in \mathcal{T}$ with minimum depth, minimizing the size.

$$\mathcal{T} = \{T_1 = \{1,2,3\}, T_2 = \{2,3,4,5,6\}, T_3 = \{2,3,4,7,8\}, T_4 = \{4,5,6,7\}\}$$

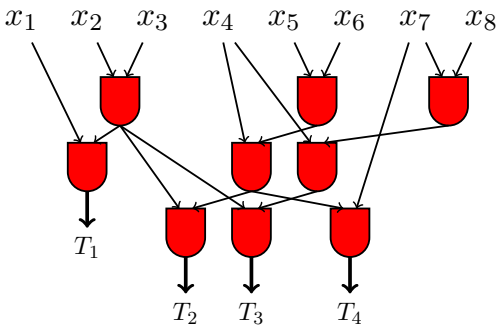


Figure 4.1: Example instance $\mathcal{T}$ with a possible implementation solving the ATSO problem. The inputs are shown as variables on the top. In red you can see the AND gates. Thick arrows pointing to $T_1, \ldots, T_4$ symbolize outputs of the circuit. They are associated with the indicated tree T.

An instance of ATSO or DATSO will usually be given by $\mathcal{T}$, implying the set of variables as union of the elements of $\mathcal{T}$. We refer to the elements of $\mathcal{T}$, for which we aim to construct a subcircuit, as trees. An example instance and a possible solution are depicted in Figure 4.1.

The ATSO problem is a special case of Multi MCSP(see Section 2.2), which has been studied in the context of computational complexity, and has been shown to be NP-hard under randomized reductions by Ilango, Loff, and Oliveira [ILO20].

However, multi-output AND functions also form an important class of functions for different applications. The fastest knows adders as presented in Chapter 3 need small enhanced AND functions to achieve linear size AND-OR paths and then also linear size adders. Our problem formulation also appears to be equivalent to the problem of minimizing the size of extended linear reformulations as investigated by Elloumi and Verchère [EV23]. Moreover, the developed algorithms can be applied to all AND components on the chip during optimization of AND-Inverter-Graphs that are used in diverse EDA applications (see Hellerman [Hel63], Darringer et al. [Dar+81], and Brayton and Mishchenko [BM10]). We will see a practical application in Chapter 5.

4.1.1 Related work

Boolean formulae and circuit size minimization are well-studied topic in complexity theory. Buchfuhrer and Umans [BU11] have been able to show that the minimization of general Boolean formulae is Σ_2^p-complete, making it an extremely challenging problem. The circuit version Multi MCSP is a well-studied topic, too. Ilango, Loff, and Oliveira [ILO20] show that the problem is NP-hard with respect to randomized reductions, improving former results which only proved this for restricted circuit classes like disjunctive normal forms (see Levin [Lev73]). Lu et al. [Lu+24] show that the problem is also hard under approximate Levin reductions. One reason to approach the Multi MCSP is that the complexity of the single-output MCSP is a longstanding open question.

For And functions, the single-output version is clearly in P, and various researchers have worked on optimizing single And or Xor trees with respect to other metrics. See Section 4.1.3 for more details on Xor functions. A single tree has a fixed size, and optimizing the delay (or depth) can easily be done by a dynamic program, as shown by Golumbic [Gol76] and others like Parker [Par79] after him. There are different approaches to optimize such trees in practice, including different further objectives, for example by Xiang et al. [Xia+10], Geisen [Gei12], and Held and Kämmerling [HK17].

And trees are often optimized as prefix graphs within adders, on which there are various results that we already discussed in Section 3.1.2. We have given an insight in And tree optimization as part of And-Or path optimization in Section 3.6. This optimization considers the DATSO problem for a special set of instances. Before adding the modification of Algorithm 3.4, every tree of the DATSO problem was consecutive, i.e., its input indices formed an interval. Using the modification, every tree is triangular with respect to the given leftist circuit, which allows for cheap solutions. In this chapter, we consider arbitrary instances instead, however.

Regarding the optimization of several And functions at once, only little has been done so far. Nöbel [Nöb21] shows that the problem of size optimization under minimum delay constraints is NP-hard, i.e., he shows that DATSO is NP-hard. We will see in Section 4.2 that this is also true without the minimum delay constraints and already for $k = 3$.

A similar problem formulation has also been studied by Elloumi and Verchère [EV23] in the context of extended linear reformulations of binary polynomial optimization problems. When considering the NP-hard problem of minimizing a polynomial over binary variables, an important task is to linearize the monomials by introducing variables representing each of them. There are different ways how to do this as for example shown by Fortet [For60] and Dalkiran and Sherali [DS16]); and doing this in a way to minimize the number of needed variables is equivalent to k-ATSO, where k is just the degree of the initial polynomial. Elloumi and Verchère [EV23] provide the same formulation but optimize, apart from size, another objective related to how well the reformulation follows the polynomial.

4.1.2 Our results

In this chapter we give some positive and negative results regarding approximation guarantees of the k-ATSO problem. We call an algorithm an α-approximation if it runs in polynomial time and returns a solution having at most α times the optimum size. We start by showing the following hardness result.

Theorem 4.1.1. *There is some* $\alpha = \frac{261}{260} > 1$ *such that there exists no* α*-approximation for* 3*-ATSO unless* $P = NP$*.*

We use a reduction of vertex cover to prove this. This reduction is not approximation preserving but still allows us to give a lower bound on the guarantee of ATSO due to the hardness of vertex cover in graphs with bounded degree. Our main result is the following collection of approximation guarantees.

Theorem 4.1.2. *There are the following approximation guarantees.*

(i) There exists a 1.212*-approximation for* 3*-ATSO.*

(ii) There exists a 1.731*-approximation for* 4*-ATSO.*

(iii) There exists a $\frac{2}{3}k$*-approximation for* k*-ATSO.*

We start by giving combinatorial algorithms for $k = 3$ and $k = 4$ each. These algorithms yield slightly worse approximation guarantee, but they might still be useful since they can usually be implemented faster than linear programming solutions. To derive the results in the theorem, we improve on the combinatorial algorithms by providing an LP relaxation. We solve the LP and apply randomized rounding by sampling vertices for the graph according to a suitable probability distribution given by the LP solution. While this can directly be applied for $k = 3$, the rounding technique only works for 4-DATSO. Thus, we establish a reduction of 4-ATSO instances to 4-DATSO instances while only losing little in the approximation guarantee. This reduction is based on sorting out trees that are not to be built depth optimally and slightly modifying the remaining instance to get a suitable partial solution.

As we will see, the natural LP relaxation has some drawbacks for large k as the integrality gap is linear in k. To derive the guarantee for arbitrary k, we give a combinatorial Greedy algorithm, mainly focussing on finding sets of trees with large intersection. If the trees in a set $\mathcal{T}' \subseteq \mathcal{T}$ share a large number of their variables, we can build this intersection from scratch and in addition use the built intermediate result for each $T \in \mathcal{T}'$. Afterward, we consider the remaining instance separately. For this instance, we can deduce that every fairly large set of trees can only have a small overlap among their variable sets. That is why the remaining instance must have a large optimum solution compared to the number of remaining trees. We will evaluate how large precisely the intersection should be and how many trees we should at least intersect.

4.1.3 Application to XOR functions

Our results can be applied to XOR functions with a slight modification. In general, AND and XOR operators behave very similarly. Both are symmetric, commutative and associative two-input operators. However, they differ in their results when given the same input twice: we have $x \wedge x \wedge y = x \wedge y$, but $x \oplus x \oplus y = y$ for Boolean variables x and y. Thus, the problems of optimizing AND functions and optimizing XOR functions are not equivalent. However, they become equivalent when restricting the solution space to circuits that for each input i and vertex v contain at most one i-v path. Under this restriction the circuit computed at node v is precisely $\bigwedge_{i \in \mathcal{I}_v} x_i$ or $\bigoplus_{i \in \mathcal{I}_v} x_i$, respectively.

Such a restriction can lead to a worse objective value as shown for example in Figure 4.2. However, all algorithms derived in this chapter compute a circuit satisfying this restriction anyway. For both $k = 3$ and $k = 4$ there is never a

$$\mathcal{T} = \{\{1,2,3,4\}, \{2,3,4\}, \{3,4\}, \{3,4,5\}, \{3,4,5,6\}, \{1,2,3,4,5,6\}\}$$

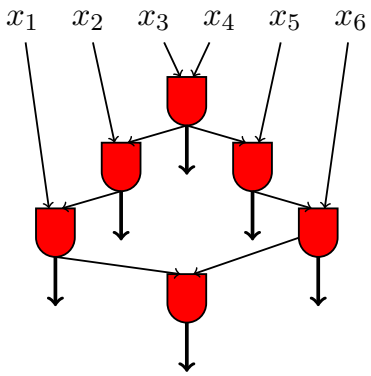


Figure 4.2: This instance has a worse solution when restricting to circuits that contain at most one i-v path per input i and vertex v. This shows the optimum solution using a general circuit. For the restricted problem at least one more gate for example computing $x_1 \wedge x_2$ is needed.

reason to build a circuit not satisfying the restriction. Whenever two predecessors with intersecting input sets are combined, the vertex only depends on four variables and thus the predecessor depending on fewer vertices can be replaced by a single variable. In the general case, one can see from the proof in Section 4.7 that such a non-restricted circuit is never built by our algorithm. Thus, all approximation guarantees claimed in Theorem 4.5.10 hold for the restricted problem, too. And, more importantly, they can be applied to the restricted problem for Xor functions defined as follows.

k-Xor Tree Size Optimization:
Instance: a set of variables $(x_1, \ldots, x_n)$ and a set of index subsets $\mathcal{T} \subseteq \mathcal{P}(\{1, \ldots, n\})$ satisfying $|T| \leq k$ for all $T \in \mathcal{T}$.
Task: find a circuit computing $\bigoplus_{i \in T} x_i$ for each $T \in \mathcal{T}$ with the minimum number of used gates such that $\mathcal{I}_{v_T} = \{x_i \mid i \in T\}$ for all $T \in T$.

The last condition that $\mathcal{I}_{v_T} = \{x_i \mid i \in T\}$ for all $T \in T$ is not strictly necessary and there are examples where this worsens the guarantee just as in Figure 4.2. Nevertheless, this condition is usually satisfied in practice, and it allows us to use our formulation for these instances, too.

Moreover, this restricted problem is also the formulation used when studying linear reformulations of binary polynomial optimization problems and the results can be applied there, too.

4.1.4 Organization of this chapter

In Section 4.2, we give a proof of the NP-hardness of ATSO and an extension to show the APX-hardness claimed in Theorem 4.1.1. For the positive results we introduce an LP formulation in Section 4.3. Section 4.4 gives two approximation algorithms for the case where each tree has at most three variables, in the end proving Theorem 4.5.10 (i). In Section 4.5, we deal with the case of four variables,

proving Theorem 4.5.10 (ii) and an additional guarantee for 4-DATSO. The LP rounding approach that works best for the cases with small k however does not work for large k due to its large integrality gap that we show in Section 4.6. Section 4.7 then shows a combinatorial algorithm for the general case, i.e., Theorem 4.5.10 (iii).

Statement of prior publication

Parts of this chapter have been previously published in concise form in Armbruster [Arm24]. This publication is due to the author of this thesis. Apart from some overlap with the introduction, it contains the result of Theorem 4.1.1, the results of the combinatorial algorithms Theorem 4.4.1 and Proposition 4.5.8, the reduction to 4-DATSO in Theorem 4.5.1 and the result for general k in Item (iii) of Theorem 4.5.10. The results related to the LP relaxation are part of this work.

4.2 Hardness of the problem

4.2.1 NP-hardness via a reduction of vertex cover

We start by giving a reduction of vertex cover to 3-ATSO to show its NP-hardness. Vertex cover has been shown to be NP-hard by Karp [Kar72].

VERTEX COVER:	
Instance:	a simple undirected graph $G = (V, E)$.
Task:	compute a minimum size subset $C \subseteq V$ of the vertices such that each edge $e \in E$ is covered, i.e., $C \cap e \neq \emptyset$.

Theorem 4.2.1. *The 3-ATSO problem is NP-hard.*

Proof. Given a vertex cover instance $G = (V, E)$ where $V = \{1, \dots, n\}$, we create the following ATSO instance. Let there be $n + 1$ Boolean variables $x_0, \dots, x_n$. For each edge $e = \{i, j\} \in E$, we create a tree $T = \{0, i, j\}$.

Claim. There is a vertex cover of size N if and only if there is a ATSO solution of size $N + |E|$ in the above construction.

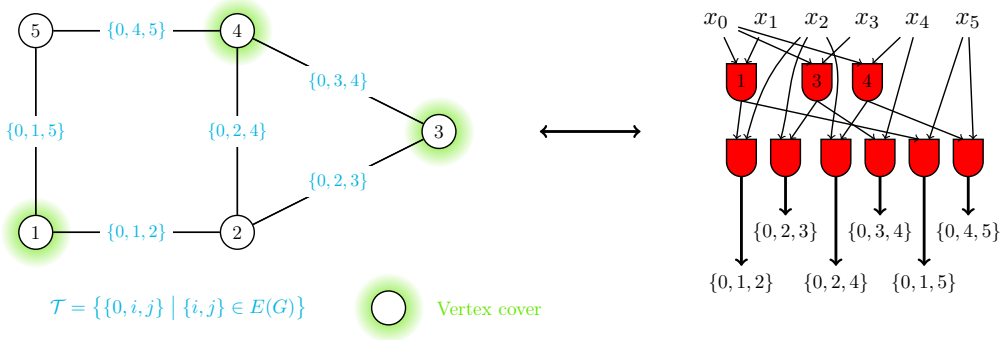


Figure 4.3: Correspondence of a vertex cover of size 3 and a solution to ATSO of size 9. Given the graph shown in black on the left, we construct the trees corresponding to the graph edges as indicated in blue. If we are given a vertex cover for the graph, we can construct a circuit as on the right and vice versa.

Proof of claim. The correspondence is illustrated in Figure 4.3. To prove this, assume you are given a vertex cover of size N. Then for each vertex i selected in the vertex cover, add a circuit node computing $x_0 \wedge x_i$, resulting in a total of N nodes. Now each $T \in \mathcal{T}$ is associated with a unique edge $\{i, j\}$ for which it was introduced. This edge is covered by one of its endpoints in the vertex cover, say without loss of generality i. Then there is already a circuit node $x_0 \wedge x_i$ and only one additional node is needed to compute $x_j \wedge (x_0 \wedge x_i)$. This makes a total of $N + |E|$ nodes.

On the contrary, suppose you are given a solution to ATSO of value $N + |E|$. There are exactly $|E|$ nodes in the circuit computing an And function on three variables. Thus, there remain N other nodes, which compute And functions on two variables as intermediate results. Since each edge $e = \{i, j\}$ requires an intermediate result for the corresponding tree $T = \{0, i, j\}$, there must be a node computing either $x_0 \wedge x_i$, $x_0 \wedge x_j$ or $x_i \wedge x_j$. If there is a node computing the last, you can replace it with a node computing $x_0 \wedge x_i$ since the result of $x_i \wedge x_j$ cannot be reused by another tree anyway. This does not change the total size of the circuit. So without loss of generality, assume that there are no nodes computing $x_i \wedge x_j$ for $i, j \geq 1$. We choose as a cover all i for which $x_0 \wedge x_i$ is computed. Then every edge is covered by one of its endpoints because it corresponds to a tree T that is computed via an intermediate result. Hence, the nodes computing And functions on two variables transform into a vertex cover. There are precisely N of these. □

This completes the proof of the NP-hardness. □

4.2.2 APX-hardness

We now show how to exploit the reduction to even get APX-hardness of ATSO, i.e., to prove Theorem 4.1.1.

Proposition 4.2.2. *If there exists no α-approximation algorithm for vertex cover in graphs of maximum degree Δ, then there does not exist a $\frac{\Delta+\alpha}{\Delta+1}$-approximation for ATSO either.*

Proof. Assume you are given a $\frac{\Delta+\alpha}{\Delta+1}$-approximation algorithm $\mathcal{A}$ for ATSO and a vertex cover instance in which the maximum degree is at most Δ. We once again consider the reduction shown in the proof of Theorem 4.2.1 and look at the ATSO instance obtained by creating one tree per edge. To solve the vertex cover instance, we solve the ATSO instance using $\mathcal{A}$ and take the vertex cover corresponding to this solution. Let alg_{VC} be the value of the solution computed thereby, and let opt_{VC} be the value of the optimum solution to the vertex cover instance. Let in addition alg_{ATSO} be the value of the solution for ATSO computed by $\mathcal{A}$ and opt_{ATSO} the optimum solution to the ATSO instance.

We observe that each vertex can cover at most Δ edges within any vertex cover. Hence,

$$\Delta \, \text{opt}_{\text{VC}} \geq |E| \ . \tag{4.1}$$

The $\frac{\Delta+\alpha}{\Delta+1}$-approximation algorithm $\mathcal{A}$ for ATSO implies that, when applied to instances generated from vertex cover,

$$
\begin{aligned}
\mathrm{alg}_{\mathrm{VC}} + |E| &= \mathrm{alg}_{\mathrm{ATSO}} \\
&\overset{\mathcal{A}}{\leq} \frac{\Delta+\alpha}{\Delta+1}\,\mathrm{opt}_{\mathrm{ATSO}} \\
&= \frac{\Delta+\alpha}{\Delta+1}(\mathrm{opt}_{\mathrm{VC}} + |E|) \\
&= \frac{\Delta+\alpha}{\Delta+1}\,\mathrm{opt}_{\mathrm{VC}} + \left(\frac{\Delta+\alpha}{\Delta+1} - 1\right)|E| + |E| \\
&\overset{(4.1)}{\leq} \frac{\Delta+\alpha}{\Delta+1}\,\mathrm{opt}_{\mathrm{VC}} + \left(\frac{\Delta+\alpha}{\Delta+1} - 1\right)\Delta\,\mathrm{opt}_{\mathrm{VC}} + |E| \\
&= \left(\frac{\Delta+\alpha}{\Delta+1} + \left(\frac{\Delta+\alpha}{\Delta+1} - 1\right)\Delta\right)\mathrm{opt}_{\mathrm{VC}} + |E| \\
&= \left(\frac{\Delta+\alpha}{\Delta+1}(\Delta+1) - \Delta\right)\mathrm{opt}_{\mathrm{VC}} + |E| \\
&= \alpha\,\mathrm{opt}_{\mathrm{VC}} + |E| \ .
\end{aligned}
$$

So a $\frac{\Delta+\alpha}{\Delta+1}$-approximation for ATSO implies an α-approximation for vertex cover. □

This allows us to prove the theorem stated in the beginning. We repeat it here for the sake of completeness.

Theorem 4.1.1. *There is some $\alpha = \frac{261}{260} > 1$ such that there exists no α-approximation for 3-ATSO unless $P = NP$.*

Proof of Theorem 4.1.1. As shown by Chlebík and Chlebíková [CC06], it is NP-hard to approximate vertex cover on 4-regular graphs within a factor of $\frac{53}{52}$. This implies by Proposition 4.2.2 that there does not exist a $\frac{261}{260}$-approximation for 3-ATSO. □

4.3 LP formulation

We introduce an LP formulation for the considered problem that we round for small values of k. For that, we use the following notation. Let $\mathcal{S} \coloneqq \{S \subseteq T \mid T \in \mathcal{T}\}$ be the set of all subsets of trees and let $\mathcal{S}_2 \coloneqq \left\{S \in \mathcal{S} \;\middle|\; |S| \geq 2\right\}$ be those that contain at least two elements. We introduce variables x_S for every $S \in \mathcal{S}$ indicating whether a node computing an AND on subset S exists. The main ingredient for the LP formulation is that every computed set S of variables needs to have a decomposition into two proper subsets that are also computed. This results in the following LP formulation:

$$
\begin{aligned}
\min \quad & \sum_{S \in \mathcal{S}_2} x_S \\
s.t. \quad & \sum_{\{S', S\setminus S'\} \,:\, \emptyset \subsetneq S' \subsetneq S} \min\{x_{S'}, x_{S\setminus S'}\} \;\geq\; x_S \qquad \forall\, S \in \mathcal{S}_2 \\
& x_T \;=\; 1 \qquad \forall\, T \in \mathcal{T} \\
& x_S \;\in\; [0,1] \quad \forall\, S \in \mathcal{S} \ .
\end{aligned}
$$

(ATSO LP relaxation)

Note that this relaxation is indeed linear as the minima only occur on the larger side of the constraint and can thus easily be modeled by a separate variable.

Moreover, it can be solved in polynomial time for every constant k. We use this LP formulation both for $k = 3$ and $k = 4$.

Note that this relaxation follows the restriction given in Section 4.1.3. But as we use this LP only in the cases $k = 3$ and $k = 4$, this does not matter at all. Moreover, it can be modified to an LP for the general problem by summing not over all partitions $(S', S \setminus S')$ but over all covers (S', S'') such that $S' \cup S'' = S$.

4.4 Approximating 3-ATSO

In this part we prove Theorem 4.5.10 (i) from the introduction, i.e., we give an efficient 1.212-approximation algorithm for 3-ATSO. We start by giving an easy $\frac{4}{3}$-approximation. We the use the LP given in Section 4.6 for our case and in the end tune the sampling probabilities we use to sample gates from our LP solution.

Observe that the case of every tree having at most three inputs can easily be reduced to the case of every tree having exactly three inputs: for every tree with two inputs, there is only one way how to build it, and for every tree with three inputs containing such a subtree with two inputs as a subset, we do not need any intermediate result. This is clearly optimum and the remaining instance is then independent. Hence, without loss of generality assume that $|T| = 3$ for all $T \in \mathcal{T}$.

4.4.1 Combinatorial algorithm

Algorithm 4.1: Combinatorial 3-ATSO algorithm

1 $\mathcal{T}_R \coloneqq \mathcal{T}$
2 **while** $\exists S = \bigcap_{i=1}^{\ell} T_i$ *such that* $T_i \in \mathcal{T}_R$ *for* $1 \leq i \leq \ell$, $\ell \geq 3$ *and* $|S| = 2$ **do**
3 Build S using one vertex.
4 Build each $T \in \mathcal{T}_R$ for which $S \subsetneq T$ from S using one additional vertex.
5 $\mathcal{T}_R = \mathcal{T}_R \setminus \{T \in \mathcal{T}_R \mid S \subsetneq T\}$
6 Build each $T \in \mathcal{T}_R$ using two vertices.

The combinatorial algorithm is summarized in Algorithm 4.1. Its main idea is that reusing an intermediate result at least three times is already pretty close to optimum. Thus, our algorithm works as follows: we iteratively take pairs of variables that appear in at least three trees of $\mathcal{T}$ that have not already been built. For such a variable pair, let $\ell \geq 3$ be the number of these trees. We build all the ℓ trees containing these two variables by adding a node computing the variable pair and ℓ nodes computing the final tree results. Thus, we need $\ell + 1$ vertices in total for one such step. We iterate this step among the remaining trees.

Once there is no variable pair left that is contained in at least three trees, the other trees are built from scratch.

Theorem 4.4.1. *Algorithm 4.1 gives a $\frac{4}{3}$-approximation for 3-ATSO.*

Proof. We apply Algorithm 4.1, which we analyze in the following. For this analysis, we refer to $\mathcal{T}_R$ as the set of trees remaining to be built in line 6. We consider an optimum solution $\mathrm{OPT}(\mathcal{T})$ with value $\mathrm{opt}(\mathcal{T})$ for all trees $\mathcal{T}$. It has at least $\mathrm{opt}(\mathcal{T}) \geq \mathrm{opt}(\mathcal{T}_R) + |\mathcal{T} \setminus \mathcal{T}_R|$ gates, where $\mathrm{opt}(\mathcal{T}_R)$ is the value of an optimum solution for $\mathcal{T}_R$. This is because $\mathrm{OPT}(\mathcal{T})$ contains some solution to $\mathcal{T}_R$ and additionally at least one gate for each tree in $\mathcal{T} \setminus \mathcal{T}_R$. Furthermore, it yields that $\mathrm{opt}(\mathcal{T}_R) \geq \frac{3}{2}|\mathcal{T}_R|$ since at most two trees contained in $\mathcal{T}_R$ can share an intermediate result as no three share a variable pair.

Our algorithm now uses $\ell+1$ gates for ℓ trees in every iteration of the while-loop, where $\ell \geq 3$. Thus, the total size of all the trees computed in the first step can be upper bounded by $\frac{4}{3}|\mathcal{T}\backslash\mathcal{T}_R|$. Let $\text{alg}(\mathcal{T})$ be the value of the solution of the algorithm. Putting everything together, we get

$$\frac{\text{alg}(\mathcal{T})}{\text{opt}(\mathcal{T})} \leq \frac{2|\mathcal{T}_R| + \frac{4}{3}|\mathcal{T}\backslash\mathcal{T}_R|}{\frac{3}{2}|\mathcal{T}_R| + |\mathcal{T}\backslash\mathcal{T}_R|} = \frac{4}{3}. \qquad \square$$

Instead of building everything from scratch in line 6, one could solve the remaining instance optimally by computing a matching among the pairs of two inputs. This would lead to a better solution on most instances. However, it does not give a better approximation guarantee. An example for this algorithm as well as this issue can be seen in Figure 4.4.

$$\mathcal{T} = \{T_1 = \{1,2,3\}, T_2 = \{2,3,4\}, T_3 = \{2,3,6\},$$
$$T_4 = \{3,5,7\}, T_5 = \{4,5,7\}, T_6 = \{5,6,7\},$$
$$T_7 = \{6,7,8\}, T_8 = \{7,8,9\}, T_9 = \{1,2,4\}\}$$

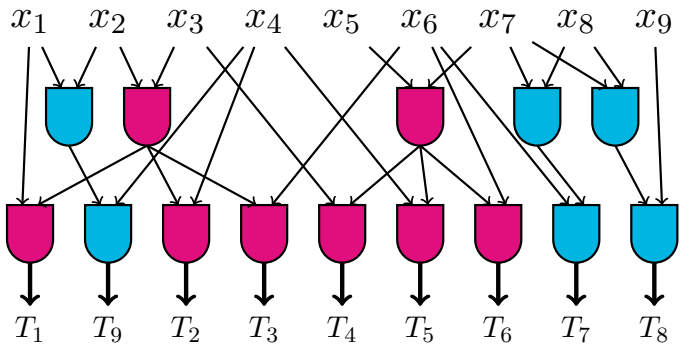


Figure 4.4: Example of Algorithm 4.1. In purple we show the gates that arise from building common pairs, in this case $\{2,3\}$ and $\{5,7\}$. In blue we show all remaining trees (in this case T_7, T_8 and T_9). Note that we twice build a vertex to compute $x_7 \wedge x_8$, which can be avoided by computing a matching on the remaining trees instead.

4.4.2 LP rounding algorithm

We use the following slightly modified but simpler LP:

$$\begin{aligned} \min \quad & \sum_{\{i,j\} \subsetneq T \in \mathcal{T}} x_{ij} \\ s.t. \quad & x_{ij} + x_{i\ell} + x_{j\ell} \geq 1 \qquad \forall T = \{i,j,\ell\} \in \mathcal{T} \\ & x_{ij} \in [0,1] \qquad \forall \{i,j\} \subsetneq T \in \mathcal{T} \ . \end{aligned}$$

(3-ATSO LP relaxation)

Let x^* be an optimum LP solution to the 3-ATSO LP relaxation and let c_{LP} be its value. Our goal is to randomly round the fractional solution to an integral one. Note that if we were allowed to have fractional solutions among the two input vertices, we could derive a solution of value $\text{opt} = c_{LP} + |\mathcal{T}|$, so this is what we compare against.

Consider the following first version of our algorithm: sample from the LP solution and build each $\{i,j\}$ with probability x^*_{ij}. Afterward, build every tree using a previously built subset if available.

The probability that a tree $T = \{i, j, \ell\} \in \mathcal{T}$ is not covered by the sampled two-input subsets is bounded by

$$\Pr[T \text{ is not covered}] = (1 - x^*_{ij})(1 - x^*_{i\ell})(1 - x^*_{j\ell}) \leq \left(\frac{3 - x^*_{ij} - x^*_{i\ell} - x^*_{j\ell}}{3}\right)^3 \leq \frac{8}{27} \ .$$

For the second inequality we have used the arithmetic-geometric mean inequality; and for the third we have used the first LP constraint. The expected cost of the sampled sets is c_{LP} and thus, the total expected cost of this solution is bounded by $c_{LP} + (1 + \frac{8}{27})|\mathcal{T}|$ which gives a $(1 + \frac{8}{27})$-approximation algorithm.

Improving the sampling probabilities

To achieve the guarantee claimed in Theorem 4.5.10(i), we improve upon this by choosing better sampling probabilities.

Theorem 4.4.2. *Algorithm 4.2 gives a randomized* 1.212*-approximation algorithm for* 3*-ATSO when given* $\beta = 1.212$*.*

Algorithm 4.2: 3-ATSO LP rounding algorithm

Input: a 3-ATSO instance, $\beta \in (0, 3)$.

1 Solve (3-ATSO LP relaxation) and get an optimum solution x^*.
2 **for** $\{i, j\} \subseteq T$ *such that* $T \in \mathcal{T}$ **do**
3 Build vertex $\{i, j\}$ with probability $\min\{\beta x^*_{ij}, 1\}$.
4 **for** $T \in \mathcal{T}$ **do**
5 Build T reusing an intermediate result if available.

Proof. Compared to the above, we scale the probabilities by a constant factor $\beta \geq 1$ and we sample every vertex $\{i, j\}$ with probability $\min\{\beta x^*_{ij}, 1\}$ instead of with probability x^*_{ij}. Let $\mathcal{U}$ be the set of sampled subsets $\{i, j\}$. Then the expected cost of our sample is

$$\mathbb{E}\big[|\mathcal{U}|\big] = \sum_{\{i,j\}} \min\{\beta x^*_{ij}, 1\} \leq \beta c_{LP} \ .$$

Moreover, the probability that a tree $T \in \mathcal{T}$ is not covered by $\mathcal{U}$ is

$$\Pr[T \text{ is not covered}] \leq \max\Big\{0, (1 - \beta x^*_{ij})(1 - \beta x^*_{i\ell})(1 - \beta x^*_{j\ell})\Big\} \leq \left(\frac{3 - \beta}{3}\right)^3 \ , \tag{4.2}$$

using again the arithmetic-geometric mean inequality. This implies that the total cost is bounded by

$$\beta c_{LP} + \left(1 + \left(\frac{3 - \beta}{3}\right)^3\right)|\mathcal{T}| \ .$$

Comparing this to the cost of the LP based fractional solution of $c_{LP} + |\mathcal{T}|$, the approximation ratio is minimum for $\beta = 1 + \left(\frac{3-\beta}{3}\right)^3$. This equation has a unique

real solution that is given by

$$\beta = 3\left(1 - \frac{1}{\sqrt[3]{\sqrt{2}-1}} + \sqrt[3]{\sqrt{2}-1}\right) ,$$

and two complex solutions in which we are not interested any further. Indeed, one can check that this solution satisfies the condition

$$\begin{aligned}
1 + \left(1 - \frac{\beta}{3}\right)^3 &= 1 + \left(1 - \left(1 - \frac{1}{\sqrt[3]{\sqrt{2}-1}} + \sqrt[3]{\sqrt{2}-1}\right)\right)^3 \\
&= 1 + \left(\frac{1}{\sqrt[3]{\sqrt{2}-1}} - \sqrt[3]{\sqrt{2}-1}\right)^3 \\
&= 1 + \frac{1}{\sqrt{2}-1} - 3\frac{1}{\sqrt[3]{\sqrt{2}-1}} + 3\sqrt[3]{\sqrt{2}-1} - \sqrt{2} + 1 \\
&= \beta \ .
\end{aligned}$$

Since $\beta < 1.212$, this implies a 1.212-approximation. □

Note that one could choose β depending on the ratio between c_{LP} and $|T|$ which would lead to better results for some ratios. However, one can check that for $\frac{c_{LP}}{|T|} \approx 0.3553$, the worst case guarantee is indeed attained. This can therefore not lead to a better approximation guarantee in general.

This allows us to prove Item (i) of Theorem 4.5.10 which we restate for completeness.

Theorem 4.5.10. *There are the following approximation guarantees.*

(i) There exists a 1.212*-approximation for* 3*-ATSO.*

(ii) There exists a 1.731*-approximation for* 4*-ATSO.*

(iii) There exists a $\frac{2}{3}k$*-approximation for* k*-ATSO.*

Proof of Theorem 4.5.10(i). The required approximation guarantee is given by Theorem 4.4.2. To conclude the proof, we need to show that this algorithm can be derandomized. We use the method of conditional expectations. More precisely, whenever we would make a random decision whether to build vertex $\{i, j\}$, we instead round this to the variant having the better conditional expectation. Let $S_1, \ldots, S_r$ be the ordered elements considered in line 3, i.e., the two-element subsets of the trees, and let p_{S_i} for $1 \leq i \leq r$ be the probability with which set S_i is built in the randomized algorithm. We iterate over the sets S_i and for every $1 \leq i \leq r$ we compute the conditional expectations of the solution value, on the one hand given that we build S_i and on the other that we do not. In every iteration, we condition on the decisions we have made in all earlier iterations. For $1 \leq i \leq r$ let A_{i-1} be the event that for all $1 \leq j < i$ the sets S_j are built as we have decided in earlier iterations. Then for $1 \leq i \leq r$, these expectations are given by

$$\begin{aligned}
\mathbb{E}_i^0 &:= \mathbb{E}\big[|\mathcal{U}| + |\{T \text{ not covered by } \mathcal{U}\}| \;\big|\; (S_i \notin \mathcal{U}) \wedge A_{i-1}\big] \\
\mathbb{E}_i^1 &:= \mathbb{E}\big[|\mathcal{U}| + |\{T \text{ not covered by } \mathcal{U}\}| \;\big|\; (S_i \in \mathcal{U}) \wedge A_{i-1}\big],
\end{aligned}$$

If $\mathbb{E}_i^1 \leq \mathbb{E}_i^0$, we build S_i, else we do not build it. Thus, if $\mathbb{E}_i^1 \leq \mathbb{E}_i^0$, we set event $A_i \coloneqq A_{i-1} \wedge (S \in \mathcal{U})$, else we set event $A_i \coloneqq A_{i-1} \wedge (S \notin \mathcal{U})$ (where A_0 is the sure event). Let additionally $\mathbb{E}_i \coloneqq \begin{cases} \mathbb{E}_i^1 & \text{if } S \in \mathcal{U} \\ \mathbb{E}_i^0 & \text{else .} \end{cases}$

As $\mathbb{E}_{i-1} = p_{S_i}\mathbb{E}_i^0 + (1 - p_{S_i})\mathbb{E}_i^1$ is a convex combination of the two expected values, it holds that $\mathbb{E}_i^0 \leq \mathbb{E}_{i-1}$ or $\mathbb{E}_i^1 \leq \mathbb{E}_{i-1}$. Thus, our choice of A_i ensures such that $\mathbb{E}_i \leq \mathbb{E}_{i-1}$. This implies the decreasing sequence

$$\mathbb{E}\big[|\mathcal{U}| + |\{T \text{ not covered by } \mathcal{U}\}|\big] = \mathbb{E}_0 \geq \mathbb{E}_1 \geq \cdots \geq \mathbb{E}_r \ .$$

As $\mathbb{E}_r$ is just the cost of the deterministic solution, and the left part is the cost of the randomized solution, this implies that the derandomization does not increase the cost.

To conclude, we need to show that these expected values can be computed in polynomial time. Let $\mathcal{U}_i$ be the sets built after iteration i, i.e., the ones that are ensured to be in $\mathcal{U}$ by A_i. The conditional expectations can be computed as

$$\mathbb{E}_i = |\mathcal{U}_i| + \sum_{j=i+1}^{r} p_{S_j} + \sum_{T=\{r,s,t\}\in\mathcal{T}} \Big(1 - p^i_{\{r,s\}}\Big)\Big(1 - p^i_{\{r,t\}}\Big)\Big(1 - p^i_{\{s,t\}}\Big);$$

where

$$p^i_{\{r,s\}} = \begin{cases} 1 & \text{if } S_j = \{r,s\} \text{ has index } j \leq i \text{ and } S_j \in \mathcal{U}_i \\ 0 & \text{if } S_j = \{r,s\} \text{ has index } j \leq i \text{ and } S_j \notin \mathcal{U}_i \\ p_{\{r,s\}} & \text{else .} \end{cases}$$

This can clearly be done in polynomial time. □

4.5 Approximating 4-ATSO

In this section we prove Theorem 4.5.10 (ii) from the introduction, i.e., that there is a 1.731-approximation algorithm for 4-ATSO. First, we give a reduction of 4-ATSO to 4-DATSO which loses only a small factor in the approximation guarantee. Afterward, we show two different algorithms for 4-DATSO: a combinatorial algorithm using a vertex cover reduction and a randomized LP rounding algorithm that achieves the necessary guarantee after choosing careful sampling probabilities. The reduction and the algorithms can then be combined to prove the desired result regarding 4-ATSO.

4.5.1 Reduction of 4-ATSO to 4-DATSO

The reduction is quantified as follows.

Theorem 4.5.1 (4-DATSO reduction)**.** *Let $\frac{4}{3} \leq \alpha \leq 2$. Given an α-approximation for 4-DATSO, there is a $\big(1 + \frac{\alpha}{2}\big)$-approximation of 4-ATSO.*

In several steps, we reduce the instance until the remaining and slightly modified instance has an optimum solution of depth 2. Then we can apply the DATSO algorithm to it. To justify that it suffices to remove trees sharing three inputs, we first prove the following lemma.

Lemma 4.5.2. *Let $\mathcal{T}$ be a 4-ATSO instance. If for all trees $T, T' \in \mathcal{T}$ we have $|T \cap T'| \leq 2$, then there is an optimum solution for ATSO that satisfies the conditions of a DATSO solution, i.e., such that every tree is built with depth at most 2.*

Proof. The reduction is shown in Figure 4.5. Since no tree can share more than two Boolean variables with any other tree, intermediate results computing a function on three inputs are never used more than once. If such a node v computes without loss of generality $\{1,2,3\}$, and the tree actually wants to compute $\{1,2,3,4\}$, the last node u of the tree combines v with x_4. Node v itself must have a predecessor w which computes an intermediate result, say without loss of generality $\{1,2\}$. Removing v and introducing a node w' computing $\{3,4\}$ instead and then computing u from w and w' does not increase the number of nodes but makes node u have depth 2. Hence, we can successively eliminate all trees of depth 3 and end up with the desired circuit of depth 2. □

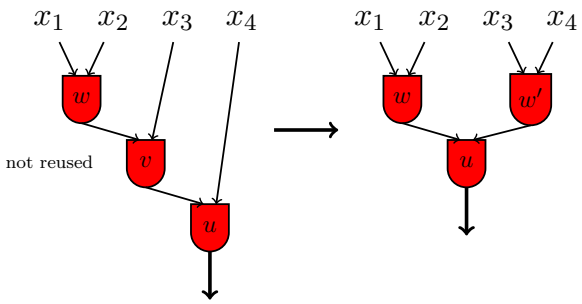


Figure 4.5: We can apply this transformation to all trees in $\mathcal{T}_4$ that are of depth 3 in the optimum solution. This does not change the optimality.

The reduction from 4-ATSO to 4-DATSO is formalized as Algorithm 4.3. Regarding notation, let $\mathcal{T}_i \subseteq \mathcal{T}$ for $i \in \{2,3,4\}$ be the set of trees that contain exactly i variables. Our goal is to get rid of trees $T \in \mathcal{T}_4$ which are cheap to build, then apply an algorithm $\mathcal{A}$ for 4-DATSO and build the cheap trees using the result of $\mathcal{A}$ in the end. The reduction steps are the following ones. First, we get rid of all trees $\mathcal{T}_4^{\supsetneq 3} \subseteq \mathcal{T}_4$ that (strictly) contain a tree $T' \in \mathcal{T}_3$. These trees T can be built easily using a single vertex by combining the result of the subtree $T' \in \mathcal{T}_3$ with the remaining input. Second, if at least $\ell \geq 3$ trees with four variables each intersect in three of their variables, we build their intersection (see lines 2 to 4). The set of trees for which we already build all needed intermediate results in this step is called $\mathcal{T}_4^{\text{intersect}}$. After this step, the remaining instance consists of $\mathcal{T}_2, \mathcal{T}_3$ and a part of $\mathcal{T}_4$ for which each triple of variables appears in at most two trees. To resolve the latter, we create a graph G defined by Equation (4.3). In this graph, we compute a maximum matching $M^{\max}$ to decide which pairs of four-variable trees shall be built sharing three variables (see line 5). However, instead of immediately building the needed five vertices for the two trees, we add their intersection to the remaining instance. More precisely, for each matching edge $\{T, T'\}$, we add $T \cap T'$ to the remaining instance while removing T and T' from it. Let the set of trees covered by the matching be $\mathcal{T}_4^{\text{matching}}$.

After these three reduction steps, the reduced instance $\mathcal{R}$ consists of

$$\mathcal{T} \cup \left\{ T \cap T' \;\middle|\; \{T, T'\} \in M^{\max} \right\} \setminus \left(\mathcal{T}_4^{\text{matching}} \cup \mathcal{T}_4^{\text{intersect}} \cup \mathcal{T}_4^{\supsetneq 3} \right) .$$

An example of how this subdivision looks like is given by Figure 4.6. We solve this instance using algorithm $\mathcal{A}$ for 4-DATSO and conclude the algorithm by building all

Algorithm 4.3: Reduction of 4-ATSO to 4-DATSO

Input: a 4-ATSO instance and an algorithm $\mathcal{A}$ for 4-DATSO.
Output: a solution to the 4-ATSO instance.

1 $\mathcal{R} \leftarrow \mathcal{T} \setminus \left\{T \in \mathcal{T}_4 \;\middle|\; \exists T' \in \mathcal{T}_3 \text{ s.t. } T' \subsetneq T\right\}$ // trees removed here form $\mathcal{T}_4^{\supsetneq 3}$.
2 **while** $\exists T_1, \ldots, T_\ell \in \mathcal{T}_4 \cap \mathcal{R}$ *s.t.* $\ell \geq 3$ *and* $|\bigcap_{i=1}^{\ell} T_i| = 3$ **do**
3 Build $S = \bigcap_{i=1}^{\ell} T_i$ with two vertices.
4 $\mathcal{R} = \mathcal{R} \setminus \{T_1, \ldots, T_\ell\}$ // trees removed in this loop form $\mathcal{T}_4^{\text{intersect}}$.
5 Compute a maximum matching $M^{\max}$ in the graph defined by

$$V(G) = \mathcal{T}_4 \cap \mathcal{R}, \qquad E(G) = \left\{\{T, T'\} \;\middle|\; |T \cap T'| = 3\right\} . \tag{4.3}$$

6 $\mathcal{R} \leftarrow \mathcal{R} \cup \left\{T \cap T' \;\middle|\; \{T, T'\} \in M^{\max}\right\} \setminus V(M^{\max})$ // trees in $V(M^{\max})$ form $\mathcal{T}_4^{\text{matching}}$.
7 Use $\mathcal{A}$ to compute a solution for $\mathcal{R}$.
8 Build all trees $T \in \mathcal{T} \setminus \mathcal{R}$ using one vertex from a subtree of T that was built by $\mathcal{A}$ or in line 3.

the trees we removed from the instance, each with a single gate reusing already built subtrees (see line 8).

For an instance $\mathcal{F}$ let $\text{opt}(\mathcal{F})$ and $\text{alg}(\mathcal{F})$ be the value of an optimum and the computed solution, respectively.

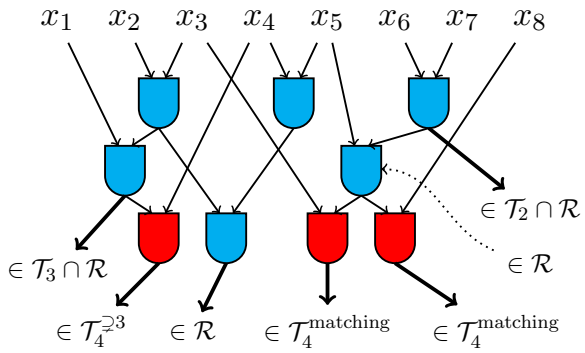


Figure 4.6: Example partition of the instance as computed when applying Algorithm 4.3.

Using this description of the algorithm we now prove the desired approximation guarantee. For this, we are going over the algorithm again, proving some quantitative results for the different steps.

Trees containing trees with three variables

In order not to lose too much size on trees that are easy to build, we need to be very careful when building the trees that share a lot of variables with each other. First, it is easy to see that trees in $\mathcal{T}_4$ that contain a tree of $\mathcal{T}_3$ need exactly one additional gate compared to a solution of the remaining instance. We build this gate just after building the remaining instance.

Lemma 4.5.3. *We have that*

$$\text{opt}(\mathcal{T}) = \text{opt}\left(\mathcal{T} \setminus \mathcal{T}_4^{\supsetneq 3}\right) + |\mathcal{T}_4^{\supsetneq 3}| \; . \tag{4.4}$$

Proof. On the one hand, the vertex computing the final result for a $T \in \mathcal{T}_4^{\supseteq 3}$ cannot be reused in any tree $T' \in \mathcal{T} \setminus \mathcal{T}_4^{\supseteq 3}$, especially not in an optimum one. This implies $\mathrm{opt}(\mathcal{T}) \geq \mathrm{opt}\left(\mathcal{T} \setminus \mathcal{T}_4^{\supseteq 3}\right) + |\mathcal{T}_4^{\supseteq 3}|$.

On the other hand, one can make every solution for $\mathcal{T} \setminus \mathcal{T}_4^{\supseteq 3}$ a solution for $\mathcal{T}$ by adding one vertex per tree in $\mathcal{T}_4^{\supseteq 3}$ that computes the final result from a tree with three inputs and the fourth input. □

Trees with large intersection

For trees with four inputs that share at least three variables with two other trees we use the following procedure. First, if there is a 3-tuple of variables that is contained in at least three four-input trees, we build a circuit for those immediately, compare Algorithm 4.3 lines 2 to 4. The 3-tuple can be computed using two gates and for each four-input tree, we need one additional gate. So we need $\ell + 2$ gates to build ℓ trees where $\ell \geq 3$. These trees are now considered built and no longer available for further tuples. We choose such 3-tuples greedily as long as there exist some. All trees chosen within this step are called $\mathcal{T}_4^{\text{intersect}}$.

Lemma 4.5.4. *We have*

$$\mathrm{opt}\left(\mathcal{T} \setminus \mathcal{T}_4^{\supseteq 3}\right) \geq \mathrm{opt}\left(\mathcal{T} \setminus (\mathcal{T}_4^{\supseteq 3} \cup \mathcal{T}_4^{\text{intersect}})\right) + |\mathcal{T}_4^{\text{intersect}}| \ . \tag{4.5}$$

Proof. The final gate computing a $T \in \mathcal{T}_4^{\text{intersect}}$ can never be reused. So removing these gates from the optimum solution for $\mathcal{T} \setminus \mathcal{T}_4^{\supseteq 3}$ still yields a solution for the remaining instance. □

After getting rid of all 3-tuples arising more than twice, we have to be careful on the remaining part. Notice that due to Lemma 4.5.4, it will be sufficient to compare to an optimum solution for the remaining tree set. This is important since we have no idea how the optimum solution for the original tree set $\mathcal{T}$ might look like. We now build pairs of trees that share three variables, compare line 5 of Algorithm 4.3. Those pairs are determined by a maximum matching algorithm to ensure that we get the maximum number of pairs.

The matching algorithm works on the graph G defined by Equation (4.3). Here, each edge $e = \{T, T'\}$ is related to a three-input set $S(e) \coloneqq T \cap T'$.

We call the set of vertices in the matching $\mathcal{T}_4^{\text{matching}}$, and we add all 3-tuples $S(e)$ we obtain from these pairs to the set of remaining trees $\mathcal{R}$.

Lemma 4.5.5. *Let $1 \leq \alpha \leq 2$. Assume you are given an α-approximation algorithm $\mathcal{A}$ for 4-DATSO. Then Algorithm 4.3 computes a $(1 + \frac{\alpha}{2})$-approximation for $\mathcal{S} \coloneqq \mathcal{T} \setminus \left(\mathcal{T}_4^{\supseteq 3} \cup \mathcal{T}_4^{\text{intersect}}\right)$.*

Proof. Let $\mathrm{OPT}(\mathcal{S})$ be an optimum solution for $\mathcal{S}$ that contains a minimum number of vertices computing functions of exactly three variables.

Our algorithm computes a maximum matching $M^{\max}$ in G as defined by Equation (4.3) and removes the trees contained in the matching. It adds the three-input sets instead and applies algorithm $\mathcal{A}$ to the computed instance and then builds the necessary additional vertices to the computed solution in order to make it a solution for $\mathcal{S}$ (see Figure 4.7).

Looking at $\mathrm{OPT}(\mathcal{S})$, there might exist some pairs of trees $T, T' \in \mathcal{S}$ whose final results are built from a common three-variable vertex $T \cap T'$. For each of those pairs,

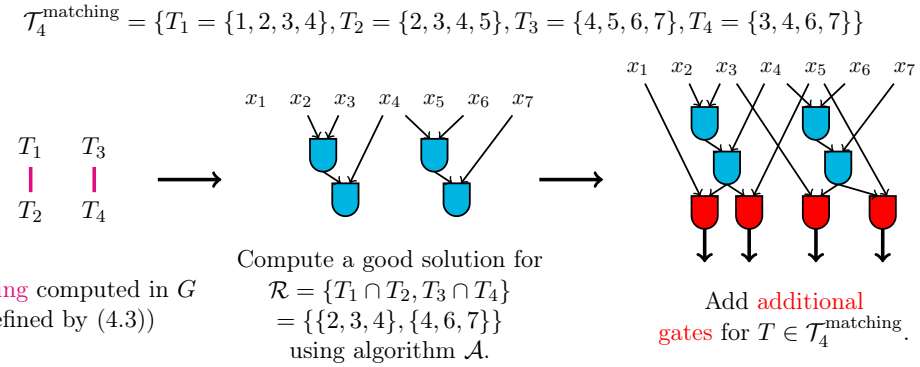


Figure 4.7: Example for the algorithm used to prove Lemma 4.5.5. $\mathcal{T}_3$ is empty in this case, $\mathcal{T}_4^{\text{matching}}$ is shown at the top.

there exists a corresponding edge in G (defined by Equation (4.3)). All these edges form a matching M^{opt} in G. Let $M^{\max}$ be the computed maximum matching in G. We consider the symmetric difference $M^{\max} \triangle M^{\text{opt}}$, i.e., the set of edges that is either in $M^{\max}$ or in M^{opt} (but not in both).

Since $M^{\max}$ is maximum, $M^{\max} \triangle M^{\text{opt}}$ consists of alternating paths and cycles, none of which is an $M^{\max}$ augmenting path. So each path starts or ends with an edge contained in $M^{\max}$. Thus, the number of inner edges belonging to $M^{\max}$ is at most the number of inner edges in M^{opt}. In the following, let each edge $e = \{T, T'\}$ be related to a three-input set $S(e) := T \cap T'$.

For each edge $e \in M^{\max}$ there are two cases: Either there exists a vertex t in the optimum solution $\text{OPT}(\mathcal{S})$ that computes a two-input subset of the three-input set $S(e)$ (compare Figure 4.8(a)). Then it is possible to build both trees $T, T' \in e$ from t using a total of three gates: one to compute the three-input set $S(e)$ and one for each T and T'. We call these edges **type 1** edges.

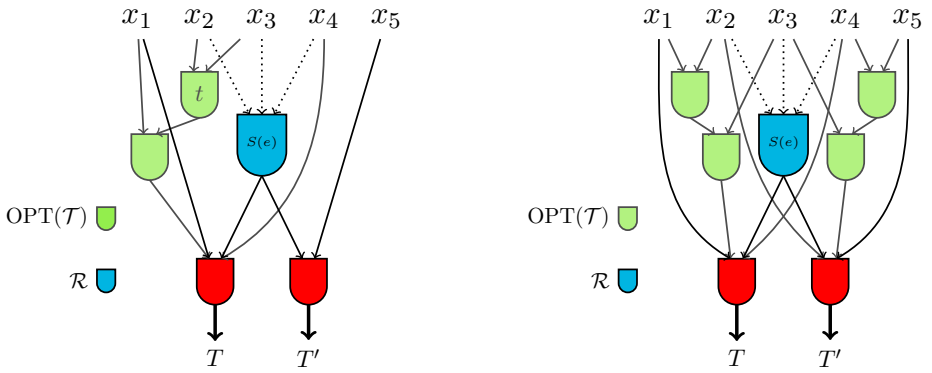
In some other cases, there might not be such a t (compare Figure 4.8(b)). This can only happen if neither of T or T' is build with depth 2 in $\text{OPT}(\mathcal{S})$ because one of the two sets that a four-input tree is built from must be contained in $T \cap T'$. In addition, the three-input functions that T and T' are built from in $\text{OPT}(\mathcal{S})$ must be different ones. Each of them must be reused by another four-input tree, otherwise we could rebuild the tree for T or T' as a depth 2 tree, contradicting our choice of $\text{OPT}(\mathcal{S})$. So speaking in terms of G, being in the second case means that e must be an inner edge of a path or cycle in $M^{\max} \triangle M^{\text{opt}}$. We call these edges **type 2** edges.

We now try to get a relation between the optimum values for $\mathcal{S}$ and $\mathcal{R}$. For this, we start with $\text{OPT}(\mathcal{S})$ and transform it such that it becomes a solution for $\mathcal{R}$. We apply the following modifications.

(i) First, we remove all final vertices computing trees $T \in V(M^{\max})$. These are $|V(M^{\max})|$ vertices.

(ii) Second, we remove all three-input vertices $S(e) = T \cap T'$ corresponding to edges of $e \in M^{\text{opt}}$ for which $T, T' \in V(M^{\max})$. These are the sets corresponding to inner edges of M^{opt} in $M^{\max} \triangle M^{\text{opt}}$ or to edges in $M^{\max} \cap M^{\text{opt}}$.

(iii) Third, for each type 2 edge e, we add one vertex to compute an intermediate



(a) Sketch of a type 1 edge in which case at least one of T and T' is built in the optimum solution using a gate that computes an intermediate result for $T \cap T'$. In this example, T is built using a gate t computing $x_2 \wedge x_3$. Now the blue gate could take as inputs the reused gate t and x_4.

(b) Sketch of a type 2 edge in which case both T and T' are built from gates computing a function on three variables in the optimum solution. Both these gates do not use a reusable gate for $S(e)$. An additional gate as intermediate result for the blue gate is needed.

Figure 4.8: Sketches of both cases distinguished in the proof of Lemma 4.5.5. In red, we can see the final gates that both the optimum and our solution for $\mathcal{T}$ contain. In green you can see the other gates of the (partial) optimum solution. In blue, you can see the tree of $\mathcal{R}$ that our algorithm wants to build instead.

result for $S(e)$. This can only be the case for inner edges of $M^{\max}$ in $M^{\max} \triangle M^{\text{opt}}$.

(iv) Last, for each edge $e \in E(M^{\max})$ we add one vertex to compute $S(e)$. These are $|M^{\max}|$ vertices.

First, note that we only removed vertices that are only used for vertices in $V(M^{\max})$, thus after step (ii), the solution is still a valid solution for $\mathcal{R} \cap \mathcal{S}$. Moreover, steps (iii) and (iv) add all the vertices needed for $\mathcal{R} \setminus \mathcal{S} = \{S(e) \mid e \in E(M^{\max})\}$, so the circuit indeed is a solution for $\mathcal{R}$.

As noted above, the number of inner edges in M^{opt} in $M^{\max} \triangle M^{\text{opt}}$ is at least the number of inner edges of $M^{\max}$ because $M^{\max}$ is a maximum matching. Thus, we add at most as many vertices in step (iii) as we deleted in step (ii). Hence, summing over all the steps implies

$$\text{opt}(\mathcal{S}) - |M^{\max}| \geq \text{opt}(\mathcal{R}) \ . \tag{4.6}$$

Furthermore, it takes exactly $2|M^{\max}|$ gates to complete a solution for $\mathcal{R}$ to a solution for $\mathcal{S}$. Using additionally that

$$|M^{\max}| = \frac{1}{2}|\mathcal{T}_4^{\text{matching}}| \leq \frac{1}{2}\,\text{opt}(\mathcal{S}) \ , \tag{4.7}$$

we get

$$\begin{aligned}
\operatorname{alg}(\mathcal{S}) &= \operatorname{alg}(\mathcal{R}) + 2|M^{\max}| \\
&\leq \alpha \operatorname{opt}(\mathcal{R}) + 2|M^{\max}| \\
&\overset{(4.6)}{\leq} \alpha\big(\operatorname{opt}(\mathcal{S}) - |M^{\max}|\big) + 2|M^{\max}| \\
&= \alpha \operatorname{opt}(\mathcal{S}) + (2-\alpha)|M^{\max}| \\
&\overset{(4.7)}{\leq} \left(1 + \frac{\alpha}{2}\right) \operatorname{opt}(\mathcal{S}) \ .
\end{aligned}$$

Thus, our algorithm has the desired approximation guarantee of $1 + \frac{\alpha}{2}$. □

Proof of Theorem 4.5.1. Putting the previous statements together, we get

$$\begin{aligned}
& \operatorname{alg}(\mathcal{T}) \\
\leq\ & \operatorname{alg}\Big(\mathcal{T} \setminus \Big(\mathcal{T}_4^{\supsetneq 3} \cup \mathcal{T}_4^{\text{intersect}}\Big)\Big) + \frac{5}{3}|\mathcal{T}_4^{\text{intersect}}| + |\mathcal{T}_4^{\supsetneq 3}| \\
\overset{\text{Lem. 4.5.5}}{\leq}\ & \left(1 + \frac{\alpha}{2}\right) \operatorname{opt}\Big(\mathcal{T} \setminus (\mathcal{T}_4^{\supsetneq 3} \cup \mathcal{T}_4^{\text{intersect}})\Big) + \frac{5}{3}|\mathcal{T}_4^{\text{intersect}}| + |\mathcal{T}_4^{\supsetneq 3}| \\
\leq\ & \left(1 + \frac{\alpha}{2}\right) \operatorname{opt}\Big(\mathcal{T} \setminus (\mathcal{T}_4^{\supsetneq 3} \cup \mathcal{T}_4^{\text{intersect}})\Big) + \frac{5}{3}|\mathcal{T}_4^{\text{intersect}}| + \frac{5}{3}|\mathcal{T}_4^{\supsetneq 3}| \\
\overset{\text{Lem. 4.5.4}}{\leq}\ & \max\left\{\frac{5}{3}, 1 + \frac{\alpha}{2}\right\}\left(\operatorname{opt}\Big(\mathcal{T} \setminus \mathcal{T}_4^{\supsetneq 3}\Big) + |\mathcal{T}_4^{\supsetneq 3}|\right) \\
\overset{\text{Lem. 4.5.3}}{=}\ & \max\left\{\frac{5}{3}, 1 + \frac{\alpha}{2}\right\} \operatorname{opt}(\mathcal{T}) \ .
\end{aligned}$$

This completes the proof. □

4.5.2 Combinatorial algorithm for 4-DATSO

We now give a combinatorial algorithm for 4-DATSO. This can be used together with the reduction of the last section to provide an algorithm for 4-ATSO.

Overview of the algorithm

Let $\mathcal{T}$ be an instance for the 4-DATSO problem. To solve the problem, we use a reduction to vertex cover within a hypergraph defined below (see Equation (4.8)).

A vertex cover in a hypergraph is just as in a usual graph a set of vertices C such that every edge contains at least one covered vertex, i.e., such that $e \cap C \neq \emptyset$ for all $e \in E(H)$. Computing a vertex cover in hypergraphs is in general even harder than in graphs. Bansal and Khot [BK10] show that assuming the Unique Games Conjecture (UGC), vertex cover in k-uniform hypergraphs is inapproximable within $k-\varepsilon$ for every $\varepsilon > 0$. Dinur et al. [Din+05] show that even when not assuming UGC one cannot approximate the problem beyond $k-1$ unless P=NP. The algorithm with the almost best known provable guarantee is the very simple algorithm of taking a maximal matching, and adding all contained vertices to a vertex cover, which in general gives a k-approximation. This is also the algorithm that we use. The only known improvements are only by a small term depending on k and possibly the maximum degree Δ of the hypergraph, see for example Halperin [Hal02].

Our hypergraph is constructed as follows.

Definition 4.5.6. Given an instance $\mathcal{T}$ for 4-DATSO, its DATSO hypergraph is given by

$$\begin{aligned} V(H) :=& \left\{ U \;\middle|\; U \subseteq T \in \mathcal{T}, |U| = 2 \right\} \\ E(H) :=& \left\{ \{U^T_{\alpha(1)}, U^T_{\alpha(2)}, U^T_{\alpha(3)}\} \;\middle|\; T \in \mathcal{T}_4, \alpha(i) \in \{i, \bar{i}\} \right\} \\ & \cup \left\{ \{U \subsetneq T\} \;\middle|\; |U| = 2, T \in \mathcal{T}_3 \right\} \\ & \cup \{T \mid T \in \mathcal{T}_2\} \ , \end{aligned} \tag{4.8}$$

where the $U^T_{\alpha(i)}$ are the two element subsets of T, i.e., let $T = \{1,2,3,4\}$ without loss of generality, then we have

$$\begin{aligned} U^T_1 &= \{1,2\} \ , & U^T_2 &= \{1,3\} \ , & U^T_3 &= \{1,4\} \ , \\ U^T_{\bar{1}} &= \{3,4\} \ , & U^T_{\bar{2}} &= \{2,4\} \ , & U^T_{\bar{3}} &= \{2,3\} \ . \end{aligned}$$

Note that this is a 3-uniform hypergraph. Its construction encodes our need to find the right depth 2 vertices as follows. The vertices of the hypergraph are pairs of indices and correspond to a pair of variables that we could combine by a gate. For each vertex chosen in the vertex cover we will build a gate combining these two variables.

For each tree $T \in \mathcal{T}_2$ we add a hyperedge that contains only the vertex consisting of this pair of variables. This is equivalent to forcing to build this pair of variables. Next, for each tree $T \in \mathcal{T}_3$, we add a hyperedge that contains all three two-element subsets of T. Thereby, one of these sets is chosen by the vertex cover algorithm, and we build one intermediate result for this tree. Last, we add hyperedges for trees with four variables. We need to ensure that two disjoint subsets are chosen as vertices in the vertex cover. Let $T = \{1,2,3,4\}$ without loss of generality. Then for each combination of one vertex of $S_1 := \{U^T_1, U^T_{\bar{1}}\} = \{\{1,2\},\{3,4\}\}$, one vertex of $S_2 := \{U^T_2, U^T_{\bar{2}}\} = \{\{1,3\},\{2,4\}\}$ and one vertex of $S_3 := \{U^T_3, U^T_{\bar{3}}\} = \{\{1,4\},\{2,3\}\}$ we create one edge containing this combination (see Figure 4.9). This ensures that if in each of these three sets S_i at least one vertex is not chosen, then there is an uncovered edge. So for one of these pairs S_i both vertices will be chosen. Figure 4.9 shows an illustration of these hyperedges for tree $T = \{1,2,3,4\}$.

The algorithm described above is summarized in Algorithm 4.4.

Algorithm 4.4: Combinatorial 4-DATSO algorithm

1. Construct the DATSO hypergraph $H = (V(H), E(H))$. Compute a vertex cover C in H as the union of all vertices of a maximal matching containing $\mathcal{T}_2$.
2. Greedily remove vertices from C that are not needed to cover $V(H)$.
3. Build a node for each U with $v_U \in C$.
4. Build each $T \in \mathcal{T} \setminus \mathcal{T}_2$ by combining vertices for U and $T \setminus U$ with one node.

Lemma 4.5.7. *Let H be a DATSO hypergraph. Every DATSO solution of value V implies a vertex cover of value $V - |\mathcal{T}_3 \cup \mathcal{T}_4|$; and every vertex cover C in H implies a DATSO solution of value at most $|C| + |\mathcal{T}_3 \cup \mathcal{T}_4|$.*

Proof. If you are given a solution of value V to $\mathcal{T}$, look at all pairs U of variables that are built in this solution and take all these as a cover C. For each $T \in \mathcal{T}_4$ there

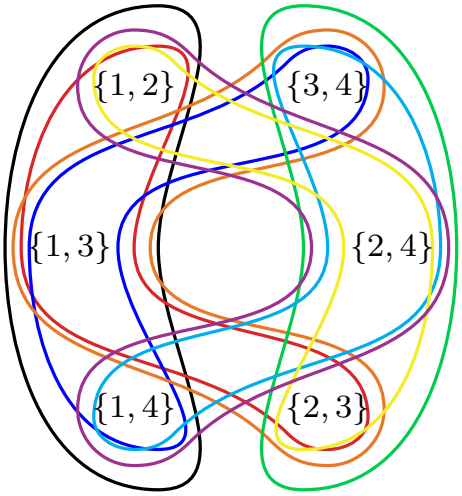


Figure 4.9: Hyperedges constructed for $T = \{1, 2, 3, 4\} \in \mathcal{T}_4$ in Algorithm 4.4. Each edge contains precisely one of the top vertices, one of the middle vertices and one of the bottom vertices.

exist sets U_i^T and $U_{\bar{i}}^T$ such that both are built in the optimum solution since T is build with depth 2. Now U_i^T and $U_{\bar{i}}^T$ together cover all hyperedges associated with T since $U_i^T \cup U_{\bar{i}}^T = T$. Similarly, for each $T \in \mathcal{T}_3$ the intermediate result covers the hyperedge corresponding to T. For each $T \in \mathcal{T}_2$ the built vertex covers the one-element hyperedge. Then $|C| = V - |\mathcal{T}_3 \cup \mathcal{T}_4|$ because exactly $|\mathcal{T}_3 \cup \mathcal{T}_4|$ nodes of the DATSO solution are used to compute functions on three or four variables.

Vice versa, consider a vertex cover C. The edges of H associated with a tree T are all covered by C. For edges arising from a $T \in \mathcal{T}_4$, this can only be true if for at least one $i \in \{1, 2, 3\}$, we have $U_i^T \in C$ and $U_{\bar{i}}^T \in C$. Otherwise, there would be at most three U^T in C (for fixed T) and then the edge containing precisely the other three U^T would not be covered. For edges arising from a $T \in \mathcal{T}_3$ covering the edge directly yields the needed intermediate result. Regarding trees $T \in \mathcal{T}_2$, this is clear, too. Hence, building nodes for each chosen U yields a set of nodes from which each T can be built using only (at most) one additional node. Thus, this implies a DATSO solution of value at most $|C| + |\mathcal{T}_3 \cup \mathcal{T}_4|$. □

Proposition 4.5.8. *Algorithm 4.4 gives a* 1.8*-approximation for 4-DATSO.*

Proof. Line 1 of Algorithm 4.4 gives us a vertex cover of size at most $3\,\mathrm{opt}_{\mathrm{VC}}$. because there a cover contains at least one vertex per matching edge and our solution uses up to three. We can be even more precise: the solution is of value at most $3\,\mathrm{opt}_{\mathrm{VC}} - 2|\mathcal{T}_2|$ because every single vertex edge is contained in our solution. We improve slightly further and ensure in line 2 that $|C| \leq |\mathcal{T}_2| + |\mathcal{T}_3| + 2|\mathcal{T}_4|$ since this is the value needed to cover every tree separately. So in fact

$$|C| \leq \min\left\{3\,\mathrm{opt}_{\mathrm{VC}} - 2|\mathcal{T}_2|, |\mathcal{T}_2| + |\mathcal{T}_3| + 2|\mathcal{T}_4|\right\} . \tag{4.9}$$

Algorithm 4.4 gives a solution of size at most $|C| + |\mathcal{T}_3 \cup \mathcal{T}_4|$ as shown in Lemma 4.5.7.

Moreover, Lemma 4.5.7 also implies that $\mathrm{opt}(\mathcal{T}) \leq \mathrm{opt}_{\mathrm{VC}} + |\mathcal{T}_3 \cup \mathcal{T}_4|$.

$$\begin{aligned}
\frac{\mathrm{alg}(\mathcal{T})}{\mathrm{opt}(\mathcal{T})} &\overset{\text{Lem. 4.5.7}}{\leq} \frac{|\mathcal{T}_3 \cup \mathcal{T}_4| + |C|}{|\mathcal{T}_3 \cup \mathcal{T}_4| + \mathrm{opt}_{\mathrm{VC}}} \\
&\overset{(4.9)}{\leq} \frac{|\mathcal{T}_3 \cup \mathcal{T}_4| + \min\{3\,\mathrm{opt}_{\mathrm{VC}} - 2|\mathcal{T}_2|, |\mathcal{T}_2| + |\mathcal{T}_3| + 2|\mathcal{T}_4|\}}{|\mathcal{T}_3 \cup \mathcal{T}_4| + \mathrm{opt}_{\mathrm{VC}}} \\
&= 1 + \frac{\min\{2\,\mathrm{opt}_{\mathrm{VC}} - 2|\mathcal{T}_2|, |\mathcal{T}_2| + |\mathcal{T}_3| + 2|\mathcal{T}_4| - \mathrm{opt}_{\mathrm{VC}}\}}{|\mathcal{T}_3 \cup \mathcal{T}_4| + \mathrm{opt}_{\mathrm{VC}}} \\
&\leq 1 + \frac{\min\{2\,\mathrm{opt}_{\mathrm{VC}} - 2|\mathcal{T}_2|, 2|\mathcal{T}| - (\mathrm{opt}_{\mathrm{VC}} - \mathcal{T}_2)\}}{|\mathcal{T}| + \mathrm{opt}_{\mathrm{VC}} - |\mathcal{T}_2|} \\
&= 1 + \min\left\{2\frac{\mathrm{opt}_{\mathrm{VC}} - |\mathcal{T}_2|}{|\mathcal{T}| + \mathrm{opt}_{\mathrm{VC}} - |\mathcal{T}_2|}, 2 - 3\frac{\mathrm{opt}_{\mathrm{VC}} - |\mathcal{T}_2|}{|\mathcal{T}| + \mathrm{opt}_{\mathrm{VC}} - |\mathcal{T}_2|}\right\} \\
&\leq 1 + \max_{x \in [0,1]} \min\{2x, 2 - 3x\} \\
&= \frac{9}{5}\ .
\end{aligned}$$

□

4.5.3 LP rounding algorithm

We improve on the previous algorithm for 4-DATSO by an LP rounding algorithm. We consider the following adapted version of the ATSO LP relaxation.

$$\begin{aligned}
\min \quad & \sum_{\{i,j\} \subsetneq T \in \mathcal{T}} x_{ij} \\
s.t. \quad & \min\{x_{ij}, x_{\ell m}\} + \min\{x_{i\ell}, x_{jm}\} + \min\{x_{im}, x_{j\ell}\} \geq 1 \quad \forall\, \{i,j,\ell,m\} \in \mathcal{T} \\
& x_{ij} + x_{i\ell} + x_{j\ell} \geq 1 \quad \forall\, \{i,j,\ell\} \in \mathcal{T} \\
& x_{ij} = 1 \quad \forall\, \{i,j\} \in \mathcal{T} \\
& x_{ij} \in [0,1] \quad \forall\, \{i,j\} \subsetneq T \in \mathcal{T}
\end{aligned}$$

(4-DATSO LP)

Note that once again, we got rid of the constant variables x_T for $T \in \mathcal{T}$. Thus, an integral LP solution is not precisely the solution to our problem but instead a set of two-variable gates to build such that they can be completed to a feasible solution without adding any intermediate vertices.

To round a solution of the LP we use an algorithm similar to before. Given some β, we modify the sampling probabilities and then build vertex $\{i,j\}$ with probability $\min\{\beta x^*_{ij}, 1\}$, where x^* is the computed optimum LP solution. Then we build every tree reusing as many vertices as possible. It is summarized as Algorithm 4.5.

Algorithm 4.5: 4-DATSO LP rounding algorithm

Input: 4-DATSO instance $\mathcal{T}$, $\beta \in (0,3)$.

1 Solve (4-DATSO LP) and get an optimum solution x^*.
2 **for** $\{i,j\} \subseteq T$ *such that* $T \in \mathcal{T}$ **do**
3 Build vertex $\{i,j\}$ with probability $p_{ij} \coloneqq \min\{\beta x^*_{ij}, 1\}$.
4 **for** $T \in \mathcal{T}$ **do**
5 Build T reusing as many intermediate results as possible.

We analyze this algorithm as before to derive the following result. Since we now need two vertices for every tree, the analysis is a bit more complex than before.

Theorem 4.5.9. *Algorithm 4.5 gives a randomized 1.462-approximation of 4-DATSO for $\beta = 1.462$. The algorithm can be derandomized.*

Proof. Analogously to Algorithm 4.2, we first solve the LP and derive a fractional solution x^*. Then we round x^* to a solution of our problem, i.e., we independently build each two-input vertex $\{i, j\}$ with a probability $p_{ij} = \min\{\beta x^*_{ij}, 1\}$. Afterward, we greedily build every tree using available intermediate results. Let $\mathcal{U}$ be the family of subsets built in line 3.

To analyze the approximation guarantee of this algorithm, we proceed similarly to Section 4.4.2. We compute the probability that a tree $T = \{i, j, \ell, m\} \in \mathcal{T}_4$ can not reuse any vertex from $\mathcal{U}$. Due to the independence of the sampling distributions for different subsets it is given by the product of the probabilities over all two-elementary subsets. It can be bounded using the arithmetic-geometric mean inequality as follows.

$$\Pr[\text{no vertex for } T] = \prod_{\{i,j\}\subsetneq T} (1 - p_{ij}) \leq \left(\frac{6 - \sum_{\{i,j\}\subsetneq T} \beta x^*_{ij}}{6}\right)^6 \leq \left(1 - \frac{\beta}{3}\right)^6$$

The last inequality follows from the first inequality of Equation (4-DATSO LP) which implies $\sum_{\{i,j\}\subsetneq T} \geq 2$ for all $T \in \mathcal{T}_4$.

Similarly, we compute the probability that it can only reuse at most one vertex from the rounded LP solution. Here, we use the inequality between arithmetic and quadratic mean.

$$\begin{aligned}
&\Pr[\text{at most one vertex for } T] \\
&\leq (1 - p_{ij}p_{\ell m})(1 - p_{i\ell}p_{jm})(1 - p_{im}p_{j\ell}) \\
&\leq \left(\frac{3 - p_{ij}p_{\ell m} - p_{i\ell}p_{jm} - p_{im}p_{j\ell}}{3}\right)^3 \\
&\leq \left(1 - \beta^2 \frac{x_{ij}x_{\ell m} + x_{i\ell}x_{jm} + x_{im}x_{j\ell}}{3}\right)^3 \\
&\leq \left(1 - \beta^2 \frac{\min\{x_{ij}, x_{\ell m}\}^2 + \min\{x_{i\ell}, x_{jm}\}^2 + \min\{x_{im}, x_{j\ell}\}^2}{3}\right)^3 \\
&\leq \left(1 - \beta^2 \frac{\left(\min\{x_{ij}, x_{\ell m}\} + \min\{x_{i\ell}, x_{jm}\} + \min\{x_{im}, x_{j\ell}\}\right)^2}{9}\right)^3 \\
&\leq \left(1 - \frac{\beta^2}{9}\right)^3 .
\end{aligned}$$

For trees $T \in \mathcal{T}_3$ it follows from Equation (4.2) that

$$\Pr[\text{no vertex for } T] \leq \left(1 - \frac{\beta}{3}\right)^3 \leq \left(1 - \frac{\beta^2}{9}\right)^3 .$$

For trees $T \in \mathcal{T}_2$ the vertex is covered almost surely.

In our solution, we need one additional vertex if a tree $T \in \mathcal{T}_3$ is not covered; one additional vertex if a tree $T \in \mathcal{T}_4$ is provided exactly one sampled vertex; and

two additional vertices if no vertex is sampled. This implies that the expected size of the solution is

$$\mathbb{E}[C] \leq \beta c_{LP} + \sum_{T \in \mathcal{T}} \big(\Pr[\text{at most one vertex for } T] + \Pr[\text{no vertex for } T]\big) + |\mathcal{T}|$$
$$\leq \beta c_{LP} + \left(1 + \left(1 - \frac{\beta^2}{9}\right)^3 + \left(1 - \frac{\beta}{3}\right)^6\right)|\mathcal{T}| \ ,$$

where the first term are the expected sampling costs, the second term is the additional vertices for bad sampling and the third term pays for the three- or four-input vertices. Thus, the approximation ratio is upper bounded by

$$\max\left\{\beta, 1 + \left(1 - \frac{\beta^2}{9}\right)^3 + \left(1 - \frac{\beta}{3}\right)^6\right\} \ .$$

To give the best guarantee, we want to choose β in a way such that this is minimized. Equations of degree 6 in general do not necessarily have a closed form for their zeros, and we are not aware that this one does. Thus, we just use the approximate minimum $\beta = 1.462$ and check that indeed

$$1 + \left(1 - \frac{\beta^2}{9}\right)^3 + \left(\frac{3-\beta}{3}\right)^6 = 1 + \left(1 - \frac{1.462^2}{9}\right)^3 + \left(\frac{1.538}{3}\right)^6$$
$$< 1 + 0.4434 + 0.0181 = 1.4614 < \beta \ .$$

Once again, this algorithm can be derandomized using the methods of conditional expectations. The proof for this is similar to the derandomization of Algorithm 4.2 in Theorem 4.5.10(i). Let $S_1, \ldots, S_r$ be the ordered elements considered in line 3, i.e., the two-element subsets of the trees, and let p_{S_i} for $1 \leq i \leq r$ be the probability with which set S_i is built in the randomized algorithm. We iterate over the sets S_i and for every $1 \leq i \leq r$ we compute the conditional expectations of the solution value, on the one hand given that we build S_i and on the other that we do not. In every iteration, we condition on the decisions we have made in all earlier iterations. For $1 \leq i \leq r$ let A_{i-1} be the event that for all $1 \leq j < i$ the sets S_j are built precisely if we have decided so in an earlier iteration. Let $\mathcal{U}$ be the set of build vertices, and let G be the set of vertices needed to complete $\mathcal{U}$ to a solution for $\mathcal{T}$ that is used by Algorithm 4.5. Then for $1 \leq i \leq r$, these expectations are given by

$$\mathbb{E}_i^0 \coloneqq \mathbb{E}\big[|\mathcal{U}| + |G| \;\big|\; (S_i \notin \mathcal{U}) \wedge A_{i-1}\big]$$
$$\mathbb{E}_i^1 \coloneqq \mathbb{E}\big[|\mathcal{U}| + |G| \;\big|\; (S_i \in \mathcal{U}) \wedge A_{i-1}\big],$$

If $\mathbb{E}_i^1 \leq \mathbb{E}_i^0$, we build S_i, else we do not build it. Thus, if $\mathbb{E}_i^1 \leq \mathbb{E}_i^0$, we set event $A_i \coloneqq A_{i-1} \wedge (S \in \mathcal{U})$, else we set event $A_i \coloneqq A_{i-1} \wedge (S \notin \mathcal{U})$ (where A_0 is the sure event). Let additionally $\mathbb{E}_i \coloneqq \begin{cases} \mathbb{E}_i^1 & \text{if } S \in \mathcal{U} \\ \mathbb{E}_i^0 & \text{else} \end{cases}$.

As $\mathbb{E}_{i-1} = p_{S_i}\mathbb{E}_i^0 + (1 - p_{S_i})\mathbb{E}_i^1$ is a convex combination of the two expected values, it holds that $\mathbb{E}_i^0 \leq \mathbb{E}_{i-1}$ or $\mathbb{E}_i^1 \leq \mathbb{E}_{i-1}$. Thus, our choice of A_i ensures such that $\mathbb{E}_i \leq \mathbb{E}_{i-1}$. This implies the decreasing sequence

$$\mathbb{E}\big[|\mathcal{U}| + |G|\big] = \mathbb{E}_0 \geq \mathbb{E}_1 \geq \cdots \geq \mathbb{E}_r \ .$$

As $\mathbb{E}_r$ is just the cost of the deterministic solution, and the left part is the cost of the randomized solution, this implies that the derandomization does not increase the cost.

To conclude, we need to show that these expected values can be computed in polynomial time. Let $\mathcal{U}_i$ be the sets built after iteration i, i.e., the ones that are ensured to be in $\mathcal{U}$ by A_i. The conditional expectations can be computed as

$$\mathbb{E}_i = |\mathcal{U}_i| + \sum_{j=i+1}^{r} p_{S_j} + \sum_{T=\{r,s,t\}\in\mathcal{T}} \left(1 - p^i_{\{r,s\}}\right)\left(1 - p^i_{\{r,t\}}\right)\left(1 - p^i_{\{s,t\}}\right) \ ;$$

where

$$p^i_{\{r,s\}} = \begin{cases} 1 & \text{if } S_j = \{r,s\} \text{ has index } j \leq i \text{ and } S_j \in \mathcal{U}_i \\ 0 & \text{if } S_j = \{r,s\} \text{ has index } j \leq i \text{ and } S_j \notin \mathcal{U}_i \\ p_{\{r,s\}} & \text{else .} \end{cases}$$

This can clearly be done in polynomial time. □

This suffices to prove Item (ii) of Theorem 4.5.10 from the introduction.

Theorem 4.5.10. *There are the following approximation guarantees.*

(i) There exists a 1.212*-approximation for* 3*-ATSO.*

(ii) There exists a 1.731*-approximation for* 4*-ATSO.*

(iii) There exists a $\frac{2}{3}k$*-approximation for* k*-ATSO.*

Proof of Theorem 4.5.10(ii). Putting Theorem 4.5.9 and Theorem 4.5.1 together yields the desired guarantee of $1 + \frac{1.462}{2} = 1.731$. □

4.6 Integrality gap of the LP formulation

In order to give some lower bounds on the approximation guarantees that can be achieved using the Equation (ATSO LP relaxation) we study the integrality gap of it.

Integrality gap for $k = 3$

First, we consider $k = 3$. We look at the following instance with 5 inputs, in which every tree contains input 0, and two inputs of $\{1, 2, 3, 4\}$. The instance $\mathcal{T}$ consists of all possible such trees.

$$\mathcal{T} = \big\{\{0,1,2\}, \{0,1,3\}, \{0,1,4\}, \{0,2,3\}, \{0,2,4\}, \{0,3,4\}\big\}$$

Without loss of generality, we can assume that every computed set contains input 0 because every other subset occurs only once. The optimum solution needs to compute $\{0, i\}$ for all $i \in \{1, 2, 3, 4\}$ but one. This gives a total value of $6 + 3 = 9$. The best fractional solution to this LP is however given by $x_{\{0,i\}} = \frac{1}{2}$ for all $i \in \{1, 2, 3, 4\}$ and $x_T = 1$ for all $T \in \mathcal{T}$. The value of this solution is $6 + 2 = 8$. This gives an integrality gap of $\frac{9}{8} = 1.125$.

Integrality gap for general k

The LP is a powerful tool for small k. For larger k, the integrality gap of this LP relaxation is linear in k though. This can be seen even for very simple instances like a single tree T with k inputs. Such an instance has an optimum integral solution of $k-1$. To contrast this, an LP solution is given by $x_S = \frac{1}{(k+2)2^{k-2}-2k-1}$ for all $2 \leq |S| \leq k-2$ and $x_S = \frac{2^{k-2}-1}{(k+2)2^{k-2}-2k-1}$ for $|S| = k-1$.

We check that this is indeed a valid solution. For every S with $2 \leq |S| \leq k-2$ the main condition is satisfied since every such set has at least one non-trivial partition. So let $|S| = k-1$. Then S has $2^{k-2}-1$ non-trivial partitions. Thus,

$$\sum_{\{S', S\setminus S'\} : \emptyset \subsetneq S' \subsetneq S} \min\{x_{S'}, x_{S\setminus S'}\} \geq (2^{k-2}-1) \cdot \frac{1}{(k+2)2^{k-2}-2k-1} = x_S \ .$$

This solution has a value of

$$\begin{aligned} c_{LP} &= 1 + k\frac{2^{k-2}-1}{(k+2)2^{k-2}-2k-1} + \frac{2^k-2k-2}{(k+2)2^{k-2}-2k-1} \\ &= 2 + \frac{2^{k-1}-k-1}{(k+2)2^{k-2}-2k-1} \\ &\leq 2 + \frac{2}{9} \ . \end{aligned}$$

Thus, the LP has an integrality gap of at least $\frac{k-1}{c_{LP}} \geq 0.45k - 0.45$. Moreover, the structure of this LP solution is very inconvenient as all variables are tiny and thus give no hint on which vertices we should use as intermediate results. This is the reason why we use a combinatorial Greedy algorithm for $k \geq 5$ in Section 4.7 instead of an LP rounding algorithm.

4.7 Approximating k-ATSO in general

Using some ideas we got from the combinatorial algorithms for the small cases, we now give an approach to the general problem. That is, we prove Theorem 4.5.10 (iii) from the introduction by analyzing Algorithm 4.6 and showing that it yields an efficient $\frac{2}{3}k$-approximation.

Algorithm 4.6: General k-ATSO algorithm.

```
1 T^⊋ ← { T ∈ T | ∃T' ⊊ T such that T' ∈ T, |T'| ≥ k/3 }
2 T = T \ T^⊋
3 for i ∈ {2, ..., |T|} do
4     while ∃S with |S| ≥ i/(3(i−1)) k and S ⊆ T for at least i trees T ∈ T do
5         Build S with |S| − 1 gates.
6         Build each T ∈ T with S ⊆ T using |T| − |S| gates and reusing the
            result of S.
7         T = (T \ {T | S ⊆ T})
8 Build each T ∈ T on its own using |T| − 1 gates.
9 Build each T ∈ T^⊋ reusing the gates of T' ⊊ T.
```

Algorithm 4.6 is a generalization of the simple algorithm presented in Section 4.4. It works as follows. In a preprocessing step, we store all those trees that can be built

cheaply from other trees in $\mathcal{T}$. We call this set $\mathcal{T}^{\supsetneq}$. Basically, we run the algorithm without them and in line 9, we build the stored trees from the main instance.

In the main part, for each number i we try to find subsets of at least i trees that overlap in a sufficiently large number of variables that depends on i. If we find such a set S, we directly build it and also build all trees containing S. Let the set of these trees be called $\mathcal{T}'$. We remove $\mathcal{T}'$ from $\mathcal{T}$ and continue. Once we do not find such an S for any i anymore, we just build each remaining tree from scratch. In the end, we build the trees in $\mathcal{T}^{\supsetneq}$ from the already existing trees.

The analysis of Algorithm 4.6 can be divided into three parts. First, we briefly look at the guarantee of the preprocessing. Second, we show that all the trees in $\mathcal{T}'$ built in line 4 to line 7 need at most $\frac{2}{3}k|\mathcal{T}'|$ gates in total, which together with $\text{opt} \geq |\mathcal{T}'|$ gives the desired guarantee, too. Last, we show that for the remaining trees of line 8, the optimum solution is so bad that building them from scratch gives the desired guarantee.

Analysis of the preprocessing

We make the following observations.

Observation 4.7.1. Building each tree of $\mathcal{T}^{\supsetneq}$ in the end takes at most $\frac{2}{3}k$ gates because it is built from another tree containing at least $\frac{1}{3}k$ variables.

Observation 4.7.2. In the optimum solution for all of $\mathcal{T}$, there is at least one vertex per $T \in \mathcal{T}^{\supsetneq}$, namely the final vertex. Since there does not remain a tree $T \in \mathcal{T} \setminus \mathcal{T}^{\supsetneq}$ which contains a tree in $\mathcal{T}^{\supsetneq}$, all these $|\mathcal{T}^{\supsetneq}|$ gates are no longer needed and $\text{opt}(\mathcal{T} \setminus \mathcal{T}^{\supsetneq}) \leq \text{opt}(\mathcal{T}) - |\mathcal{T}^{\supsetneq}|$.

Putting these two observations together yields that we have a $\frac{2}{3}k$-approximation for the preprocessed trees. Thus, it suffices to analyze the approximation guarantee of the main part of the algorithm with respect to the restricted instance $\mathcal{T} \setminus \mathcal{T}^{\supsetneq}$

Upper bounding the computed solution

We prove that we have a $\frac{2}{3}k$-approximation for the trees that we build in lines 4 to 7.

Lemma 4.7.3. *The number of gates used to build a set of trees $\mathcal{T}'$ in lines 4 to 7 is at most $\frac{2}{3}k|\mathcal{T}'|$.*

Proof. Let $\mathcal{T}'$ be the set of trees built in such an iteration, and let $S \coloneqq \bigcap_{T \in \mathcal{T}'} T$ be their intersection. We directly build this intersection S and then reuse it for all the trees $T \in \mathcal{T}'$. We chose a threshold of $\frac{i}{3(i-1)}$ for the size of S. Since $|\mathcal{T}'| \geq i$, it is $|S| \geq \frac{i}{3(i-1)}k \geq \frac{|\mathcal{T}'|}{3(|\mathcal{T}'|-1)}k$. Thus, the number of used gates for this is bounded by

$$\begin{aligned} |S| - 1 + |\mathcal{T}'|(k - |S|) &\leq |\mathcal{T}'|k - (|\mathcal{T}'| - 1)|S| \\ &\leq |\mathcal{T}'|k - (|\mathcal{T}'| - 1)\frac{|\mathcal{T}'|}{3(|\mathcal{T}'| - 1)}k = \frac{2}{3}|\mathcal{T}'|k \ . \quad \square \end{aligned}$$

Given that $\text{opt}(\mathcal{T} \setminus \mathcal{T}') \leq \text{opt}(\mathcal{T}) - |\mathcal{T}'|$, it suffices to prove the guarantee for the remaining instance $\mathcal{T} \setminus \mathcal{T}'$.

Lower bounding the optimum solution

It is left to analyze the size of the optimum solution for the remaining instance. We show that the optimum solution cannot be too cheap, implying that any solution already satisfies the wanted guarantee.

For a vertex v, let $a(v)$ be the number of trees that use vertex v in the optimum solution, where we say that v is used by T if there is a v-v_T path for the output v_T computing T. We introduce a function defining how much of a vertex is owned by which tree. First, for gates v that correspond to variable sets $S(v)$ with less than $\frac{1}{3}k$ variables, we assign v completely to $S(v)$ if $S(v) \in \mathcal{T}$ or to no tree else. For gates v with larger variable set, we assign v by a fraction of $\frac{1}{a(v)}$ to the trees T that use v. Thereby, each vertex is assigned at most once in total. These assignments are illustrated by Figure 4.10.

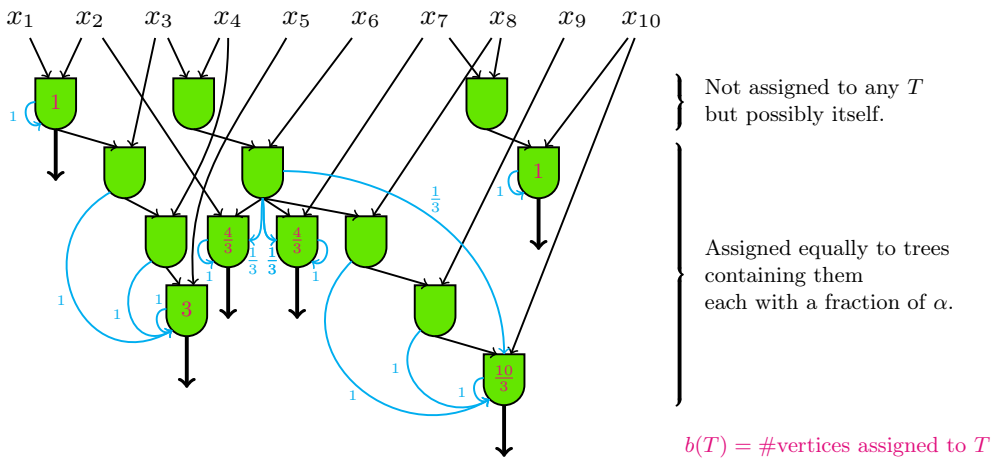


Figure 4.10: Example instance for the analysis of the second part. In blue you can see which vertex is assigned to which tree by which fraction. These numbers are basically the $\alpha(v)$ defined below. In purple the total amount $b(T)$ of gates assigned to a tree T is shown (as a number within the output vertex).

Let $b(T)$ be the total amount of fractional gates that is assigned to T. Then we have

$$\text{opt}(\mathcal{T}) \geq \sum_{T \in \mathcal{T}} b(T) \ . \tag{4.10}$$

Let in addition $\alpha(v) \coloneqq \frac{1}{a(v)}$ for gates v representing sets of size at least $\frac{1}{3}k$ and $\alpha(v) \coloneqq 0$ else. So $\alpha(v)$ is the amount of v that is assigned to each T strictly containing v.

Lemma 4.7.4. *For each tree T left in $\mathcal{T}$ in line 8, we have $\frac{|T|}{b(T)} \leq \frac{2}{3}k$.*

Proof. Let v and w be the predecessor gates of the final gate of a tree $T \in \mathcal{T}$. If either $\alpha(v) = 1$ or $\alpha(w) = 1$, then $b(T) \geq 2$ and thus, the statement is clear. The same holds if $|T| \leq \frac{2}{3}k$ since $b(T) \geq 1$ for all T due to line 1 of Algorithm 4.6.

Else, observe that $b(T) \geq 1 + \alpha(v) + \alpha(w)$. In addition, we know that

$$|T| < \frac{1}{3(1-\alpha(v))}k + \frac{1}{3(1-\alpha(w))}k$$

since otherwise, v or w would have been build in lines 4 to 7. Thus,

$$\frac{|T|}{b(T)} \leq \frac{\frac{1}{3(1-\alpha(v))}+\frac{1}{3(1-\alpha(w))}}{1+\alpha(v)+\alpha(w)}k \overset{(4.13)}{\leq} \frac{2}{3}k \ . \tag{4.11}$$

The second inequality is a computation that holds for all $\alpha \leq \frac{1}{2}$ proven as Lemma 4.7.6. □

Lemma 4.7.5. *Building everything from scratch in line 8 uses at most $\frac{2}{3}k \cdot \mathrm{opt}(\mathcal{T})$ gates.*

Proof. Summing Lemma 4.7.4 over all trees left we get

$$\begin{aligned}\frac{\sum_{T\in\mathcal{T}}(|T|-1)}{\mathrm{opt}(\mathcal{T})} &\overset{(4.10)}{\leq} \frac{\sum_{T\in\mathcal{T}}|T|}{\sum_{T\in\mathcal{T}} b(T)} \\ &\overset{(4.12)}{\leq} \max_{T\in\mathcal{T}} \frac{|T|}{b(T)} \\ &\overset{(4.11)}{\leq} \frac{2}{3}k \ .\end{aligned}$$

For the second step we used the well-known inequality

$$\frac{\sum x_i}{\sum y_i} \leq \max \frac{x_i}{y_i} \tag{4.12}$$

for positive numbers x_i, y_i. A proof can be found below in Lemma 4.7.7. This concludes the proof of the second part, i.e., that line 8 gives a $\frac{2}{3}k$-approximation for the remaining instance. □

It remains to prove the following technical ingredients.

Lemma 4.7.6. *Given two numbers $\alpha(v), \alpha(w) \in \left[0, \frac{1}{2}\right]$, we have*

$$\frac{\frac{1}{3(1-\alpha(v))}+\frac{1}{3(1-\alpha(w))}}{1+\alpha(v)+\alpha(w)} \leq \frac{2}{3} \ . \tag{4.13}$$

Proof. For every $\alpha \in \left[0, \frac{1}{2}\right]$ it holds that

$$\begin{aligned} & 0 \leq \alpha - 2\alpha^2 \\ \iff \quad & 1 \leq 1 + \alpha - 2\alpha^2 \\ \iff \quad & 1 \leq (1+2\alpha)(1-\alpha) \\ \iff \quad & \frac{1}{1-\alpha} \leq 1 + 2\alpha \ . \end{aligned} \tag{4.14}$$

Adding Equation (4.14) for $\alpha = \alpha(v)$ and $\alpha = \alpha(w)$ yields

$$\begin{aligned} & \frac{1}{1-\alpha(v)} + \frac{1}{1-\alpha(w)} \leq 2(1+\alpha(v)+\alpha(w)) \\ \iff \quad & \frac{1}{3(1-\alpha(v))} + \frac{1}{3(1-\alpha(w))} \leq \frac{2}{3}(1+\alpha(v)+\alpha(w)) \ . \end{aligned}$$

Dividing by $1+\alpha(v)+\alpha(w)$ yields the desired result of Equation (4.13). □

Last, we used the following well-known inequality, which we prove for the sake of completeness.

Lemma 4.7.7. *For non-negative real numbers $x_1, \dots, x_n$ and positive numbers $y_1, \dots, y_n$ we have*

$$\frac{\sum_{i=1}^{n} x_i}{\sum_{i=1}^{n} y_i} \leq \max_{i=1}^{n} \frac{x_i}{y_i} \ . \tag{4.15}$$

Proof. Let $M = \max_{i=1}^{n} \frac{x_i}{y_i}$. So $x_i \leq M y_i$. Summing this over all i we get

$$\sum_{i=1}^{n} x_i \leq M \sum_{i=1}^{n} y_i \ ,$$

which is equivalent to the desired inequality. □

Combining the bounds

We combine the results from the last three subsections to derive an algorithm implying Theorem 4.5.10 (iii).

Theorem 4.7.8. *Algorithm 4.6 is a $\frac{2}{3}k$-approximation for k-ATSO.*

Proof of Theorem 4.5.10 (iii). To prove the theorem, it remains to put everything together.

$$\begin{aligned}
\mathrm{alg}(\mathcal{T}) &\overset{\textit{Obs. } 4.7.1}{\leq} \mathrm{alg}(\mathcal{T} \setminus \mathcal{T}^{\supsetneq}) + \frac{2}{3}k|\mathcal{T}^{\supsetneq}| \\
&\overset{\textit{Lem. } 4.7.3}{\leq} \mathrm{alg}(\{T \text{ built in l. 8}\}) + \frac{2}{3}k|\{T \text{ built in l. 4 to 7}\}| + \frac{2}{3}k|\mathcal{T}^{\supsetneq}| \\
&\overset{\textit{Lem. } 4.7.5}{\leq} \frac{2}{3}k\,\mathrm{opt}(\mathcal{T})
\end{aligned}$$

Thus, we have shown that Algorithm 4.6 gives an overall $\frac{2}{3}k$-approximation. □

Chapter 5

And and Xor functions in practice

The problem described in Chapter 4 is highly motivated by its application in VLSI design. Thus, we dedicate this chapter to study the ATSO problem in practice. As noted in Section 4.1.3, the optimization of Xor functions is very similar to the optimization of And functions. Our approach applies for both And and Xor functions. Nevertheless, we stick to the term ATSO for describing the problem for simplicity.

The Research Institute for Discrete Mathematics at the University of Bonn maintains a close cooperation with IBM to work on practical approaches for chip design. This long-term cooperation results in the development of the software suite BonnTools, which is used by IBM in their design flow to develop server and ASIC chips. Korte, Rautenbach, and Vygen [KRV07] and Held et al. [Hel+11] have given insights into this project, and since then, it has developed further and further. Hundreds of chips have been designed using different parts of the BonnTools for different design steps.

The BonnTools contain different tools for many of the most important subproblems arising in the design flow. Apart from placement, routing and other tools, there are different tools for timing improvements, including tools for restructuring logic. One main logic restructuring routine implements a version of the And-Or path restructuring similar to the algorithms described in Chapter 3. It is used to improve timing on the most critical paths of the chip. Apart from restructuring And-Or paths, the BonnTools also contain a tool for restructuring single-output And and Xor functions. This is important after And-Or path restructuring because the splitting algorithm as described in Chapter 3 results in large And and Or trees. However, it is not only useful for that specific application but also in general as chips contain a large amount of And, Or and sometimes also Xor trees. Depending on the criticality, different objectives for optimization are used.

In this chapter we start by giving an introduction to the components and properties of a typical chip and to optimization strategies for it. In Section 5.2 we continue with an overview about the state of the art of BonnLogic prior to this work. The previous BonnLogic contains some algorithms that we reuse for our new routine. In Section 5.3 we describe how to collect and define the instances we optimize. Section 5.4 gives some details on how to simplify the instances to improve the running time of the integer linear programming solver. In Section 5.5 we describe how to put

the computed solution back into a comparable state on the chip. This includes a technology mapping, the restructuring of the contained AND or XOR trees using the existing tools, a placement and a buffering. Last, we show different examples and results of our new flow in Section 5.6.

The implementation of the instance collection described in Section 5.3 was partially implemented by Roxana Mittelberg, who was supervised by the author within the scope of this thesis.

5.1 Introduction to chip design

A typical modern chip has a rectangular chip area and consists of different physical layers. The bottom layer is called the **placement layer** and is used to place basic building blocks called cells. Most cells are logic cells, and they implement a Boolean function on their predecessors. Every chip has a couple of **primary inputs and outputs** which are connection points to the outside of the chip. A primary input provides a Boolean input variable. Primary outputs each compute a Boolean function that depends on the input variables and possibly on some bits stored from earlier iterations. To compute these functions, the chip uses a circuit as introduced in Definition 2.1.9. Every vertex of the circuit is implemented by a cell consisting of multiple transistors. Every such cell has an underlying gate type. The most common gates are NAND gates, NOR gates and inverters, but there are many more gates computing also slightly more complicated functions. Depending on the technology a chip is built in, there are different sets of gate functions available. We call the set of available gate functions the **library**. Every cell has different connection points that we call **pins**.

The higher layers are called **routing layers**, and they are used to connect the pins, including primary inputs and outputs, with each other. Every cell has some input pins and usually one output pin. Every net connects an output to some input pins and thereby transports a binary signal from one cell to its successors. We call the signal starting point the **source** of a net and the successors the **sinks**.

When propagating a signal through the chip, every gate needs to wait for the predecessor signals to arrive. Once all the signals arrived, it takes a certain time to compute the implemented function. Moreover, it also takes a certain time to propagate the computed signal from one gate to its successors. This wire delay depends on various properties of the net, including the distance and the layer that the net is on. To compute the exact realistic delay of a signal on the chip is extremely difficult as it depends on so many different properties. Some of them are internal like the width and distance to adjacent nets but others are external like the temperature for example. One can say nevertheless that the delay grows about quadratically with the traversed distance. This is one reason why nets on chips are buffered, i.e., a couple of repeaters are inserted which receive the signal and either send the same or the inverted signal anew. Another reason is that the delay grows with the number of sinks a net has. Gates that repeat the same signal, i.e., compute the identity, are called **buffers**. Gates that invert the signal are called **inverters**. As buffers are usually much larger than inverters, one often uses two inverters instead of one buffer. To model the delay, there are different models in different stages of the optimization flow that have a growing accuracy. Before a reasonable placement exists, a zero wire length model is consulted. In this model, there are only gate delays and bifurcation delays that depend on the sinks a net has. It is the model that we use for our tests because after a change of the logic cells, a new placement is

needed anyway. Moreover, our objective, the size of the circuit, does not depend on the placement, which makes it reasonable to apply our routine before the placement. Thus, this is the timing model to use in our case. After placement the timing is usually computed using a linear delay model. This is used to model that buffering in later stage usually linearizes the timing and thus, the linear delay model is a good prediction of the behavior after buffering. After buffering we compute the delay according to the so-called Elmore delay model due to Elmore [Elm48], which takes resistance and capacitance values into consideration. There are many more timing models for static timing analysis as for example presented by Rubinstein, Penfield, and Horowitz [RPH83], Pillage and Rohrer [PR90], and Ratzlaff and Pillage [RP94], but we do not discuss them here in further detail.

Timing is an important objective because every chip has a cycle time and every path on the chip needs to have finished its computation within this cycle time. To measure timing we define different properties. The **arrival time** of a pin is the time it takes until the signal arrives at this pin. It can be computed by propagating it over the chip in topological ordering. For every input pin, the arrival time is computed as the sum of the predecessor arrival time and the wire delay. For an output pin, the arrival time is the sum of the maximum input arrival time and the (possibly pin dependent) gate delay. We define the arrival time of a cell as the arrival time at its output pin. To determine whether an arrival time is critical, one also needs to take the descendants into consideration. Thus, we also introduce a **required arrival time** at the primary outputs that is the time when the signal should arrive at the latest. Analogously to the arrival time, the required arrival time can be propagated backwards from the primary outputs, i.e., in inverse topological ordering. In that way, every cell has an arrival time and a required arrival time. The difference of required and actual arrival time is called **slack**. If the slack is negative, the timing still needs to be improved.

The timing is highly influenced by the placement of the cells on the chip due to the wire delay caused between the locations of the cells. Once the cells are placed, every cell has a location and the net connects the source location to the sink locations. The length of this net is estimated by the length of a minimum Steiner tree. Building a Steiner tree saves some wiring compared to building single connections for every pair of source and sink. Total netlength is an important objective on the chip. A high netlength leads to high congestion on the chip, enforcing the nets to take detours to avoid congestion hotspots. This increases the net's delay. Moreover, lots of wiring also causes a higher power consumption. Thus, apart from timing and size, netlength is also an important objective on placed instances.

5.2 Previous work on BonnLogic

5.2.1 Overview of the tools of BonnLogic

BonnLogic is a collection of restructuring tools that are part of the BonnTools. Its first version is due to Werber, Rautenbach, and Szegedy [WRS07] and it has been developed and extended since then. It consists of different components that are called during the design flow. It contains an implementation of And-Or path restructuring similarly to Algorithm 3.1. This is applied iteratively on the most critical path. The critical path gets subdivided into subpaths consisting of logic cells only. Every subpath is transformed into an And-Or path by applying De Morgan rules (see Observation 2.1.6 Item (iv)) to move all inversions to the inputs. Then we apply a version of Algorithm 3.1 to restructure the And-Or paths, generalized

to use input delays and to allow for side-outputs. Afterward, the path it is mapped back to gates that are available on the chip like described in Section 5.5. The best subpath restructuring is implemented and the algorithm is iteratively applied to the next worst path. More details on this implementation are given by Nöbel [Nöb21] and Brenner and Silvanus [BS23] for example.

Additionally, BONNLOGIC contains a standalone tool restructuring single-output AND and XOR trees, including those that arise within the AND-OR path restructuring. In Section 3.6 we showed how to linearize the latter's size using a leftist circuit. Due to the criticality of these AND-OR path instances, linearizing the size is not the most important objective in AND-OR path restructuring. Instead, it is more important to build these trees with a minimum delay. More details on this AND and XOR tree restructuring routine can be found in the next subsection.

As further tool, BONNLOGIC contains a pin swapping routine that swaps the nets attached to equivalent pins of a single gate. Its objective is to improve netlength and timing.

Last, BONNLOGIC contains a technology mapping routine that can be applied as a standalone tool for remapping or as a mapping tool after AND-OR path restructuring or AND tree optimization. More details on this can be found in Section 5.5.

5.2.2 Previous restructuring of AND and XOR trees

Prior to this work, the BONNTOOLS only optimized single output AND and XOR functions. A description of the development of this optimization routine can be found in Geisen [Gei12]. There are further approaches by Kämmerling [Käm12] and Göke [Gök14] for the same problem, which turned out to work worse though. The main objectives during this restructuring are timing and netlength.

For this tree optimization tool an instance consists of a set of inputs on which a single output AND or XOR function needs to be computed. As input parameter the user can choose a tradeoff between the arrival time of the single output on the one hand, and the used netlength on the other hand. The main variance in netlength arises within the nets attached at the inputs because they might have some side outputs. Due to these side outputs, the netlength of an input driven net highly depends on the location of the first other input that this input is combined with.

To apply the underlying optimization algorithms, an instance collection runs over the chip in inverse topological ordering to collect the set of maximal AND and XOR trees. By skipping gates that are already part of another tree, we ensure that no gate is part of two instances. When considering a gate as a root, we start at this gate and collect a tree whose tree type is either AND or XOR depending on the gate type of the root. We collect all its ancestors as long as their gate type after applying De Morgan rules (see Item (iv) of Observation 2.1.6) coincides with the tree type. Thus, whenever we arrive at an ancestor of the root, we apply De Morgan rules to move possible inversions to the inputs of the gate and then determine whether this gate can be part of the tree. This is the case if and only if it has exactly one successor and its gate type coincides with the tree type. If so, we add it as a gate to the instance, and consider its predecessors recursively. Else, we do not consider predecessors of this gate, save this gate as an input with the suitable inversion in the circuit and move on to consider predecessors of other gates that have not yet been considered. The described procedure is captured as Algorithm 5.1.

Once all instances are collected in that fashion, we can optimize each of them separately. Given an instance, we apply a Best-Of-Many algorithm which tries different algorithms and then takes the best with respect to a given tradeoff parameter.

Algorithm 5.1: And or Xor tree collection

Input: a circuit G, a root r and an operator $\circ \in \{\text{And}, \text{Xor}\}$.
Output: the maximum $\circ$-tree T with root r.

```
1  S ← {r}
2  T ← ∅
3  while S is not empty do
4      Remove a vertex v ∈ S from S.
5      if v has exactly one successor and has gate type ∘ after applying De
         Morgan rules then
6          Add v to T as an inner vertex.
7          Add the predecessors of v to S.
8      else
9          Add v to T as an input.
10 return T
```

The algorithms to restructure the tree include a minimum Steiner tree algorithm, a bi-criteria algorithm that is a modified version of the algorithm suggested by Held and Rotter [HR13], two Greedy algorithms including the algorithm of Bartoschek et al. [Bar+06] and Huffman coding as suggested by Golumbic [Gol76].

The algorithm that turns out to be the most successful is a two-level Greedy algorithm introduced by Geisen [Gei12]. The other algorithms are used on some small fractions of the instances, too though.

After having optimized the trees, we apply technology mapping to map the constructed circuit of Ands or Xors back to the available library. More details on the technology mapping routine will be given in Section 5.5. If the And tree optimization routine is applied to buffered instances, we additionally apply buffering to the incident nets. Since the buffering takes most of the running time, we can afford to try many different fast algorithms for And or Xor tree restructuring as described above. Once the optimization is done, we compare the computed solution to the initial solution, both with respect to timing and with respect to netlength. We only keep the solution if it is better with respect to the given tradeoff.

Depending on the parameter settings this restructuring is applied either to the most timing-critical trees, to all trees on the chip or to a set of chosen trees.

5.3 Collection of instances

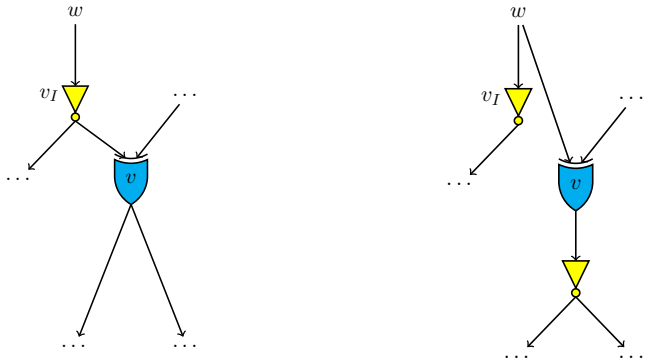
We extend the optimization of And and Xor functions using the results of Chapter 4. To apply them to a chip, the first task is to find suitable instances. On the one hand, we do not want to lose room for improvements and therefore, whenever two And functions share some inputs, we want them to be part of the same instance. On the other hand, we do not want an instance to be the union of two independent instances because that increases running times and reduces the possible running time gain that could be achieved by parallelization. Note that a parallelization is not yet implemented though. We introduce a procedure to determine the best such set of instances and elaborate why this is a reasonable choice. We start by preprocessing the netlist before we show how to collect subcones in this preprocessed netlist, and in the end, we merge related subcones to instances.

5.3.1 Preprocessing the netlist

Initially on a chip, there might not directly be large amounts of AND or XOR cells but the contained logic is mapped to a library consisting of various gate types. Thus, we to decompose all gates into 2-ary gates and inverters and apply DeMorgan rules in order to reach a state with a maximum amount of AND and XOR gates.

In a preprocessing step, we replace all gates by AND2, XOR2, and inverter gates. A representation by these three gate types is not unique of course. As of now, we represent XOR and XNOR gates using XOR2s and inverters; and represent all other available gate types using AND2s and inverters. Optimizing XORs cannot be done easily if they are represented by ANDs and inverters. Thus, we maintain them in XOR representation. Hence, in the following we only work with AND2, XOR2 and inverter gates.

We want to avoid having inverters as predecessors of XOR gates. The reason for this is that we do not want to have inverters within our instances, and thus, we move as many as possible to the border of our XOR instances.



(a) One inverter before an XOR gate.

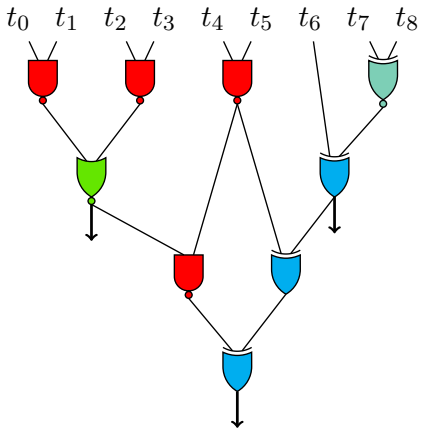
(b) Inverter moved behind the XOR gate.

Figure 5.1: Strategy to remove inverters in between two XOR gates without violating the computed logic functions.

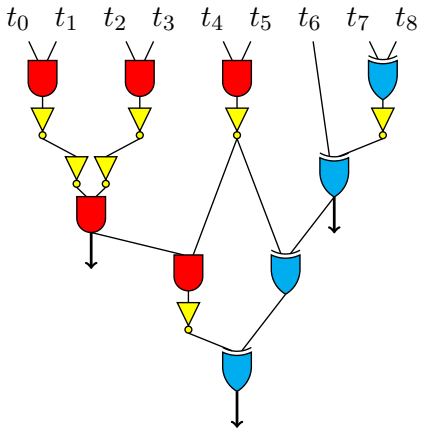
Lemma 5.3.1. *Let C be a circuit over AND2, XOR2, and inverter gates. It is possible to compute in linear time an equivalent circuit C' with the following properties. In C' no inverter is succeeded by an XOR gate, and C' contains the same number of 2-ary gates as C.*

Proof. We apply the following transformation. Whenever we find such an inverter v_I that is the predecessor of an XOR gate v, we replace it using the construction shown in Figure 5.1. That is, we replace the edge (v_I, v) by the edge (w, v), where w is the predecessor of v_I and add an inverter in between v and its successors. Since $\overline{x_1} \oplus x_2 = \overline{x_1 \oplus x_2}$, this does not change the logic. We can move (or copy) all inverters in topological order through every succeeding XOR gates. Moreover, we cancel every pair of two consecutive inverters where one is the only successor of the other. This procedure has a running time linear in the number of edges of our circuit. Moreover, it creates at most a linear number of inverters because every gate is succeeded by at most one inverter. □

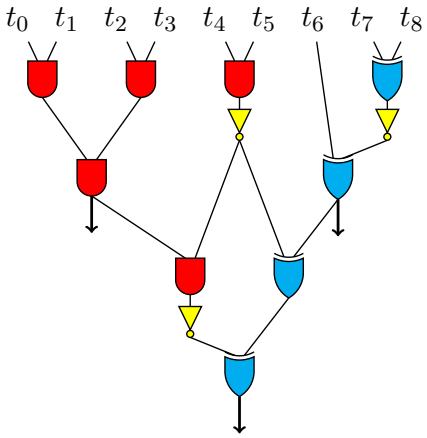
This procedure maintains the logical correctness unfortunately only for Xor operations and not for And operations, so we cannot do a similar step for And gates.



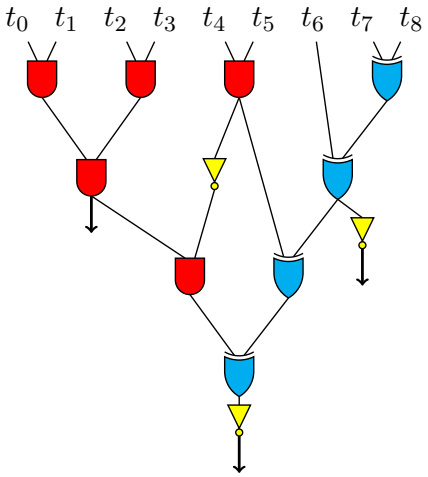
(a) Example for an instance that might be found on the chip.



(b) Decomposition of the gates into Ands, Xors and inverters.



(c) Circuit after removing pairs of consecutive inverters. This can always be done if the first inverter does not have any other sucessors.



(d) Remaining circuit after moving all the inverters through the Xor gates, and cancelling two of the three inverters that were moved to the last output.

Figure 5.2: Illustration of the preprocessing. In several steps, the instance gets changed from the state on the chip to a circuit based on And and Xor gates and inverters as shown in Figure 5.2(d). The circuit will be decomposed into three instances. The gate computing $t_4 \wedge t_5$ forms one instance, the other And gates form a second instance and all Xor gates form the last instance.

Last, we do the following preprocessing step. We rip out all inverters and reinsert at most one per net, only when necessary for logical correctness. As two succeeding inverters compute the identity function, this is always possible. For a vertex v

let $w(v)$ be a predecessor when looking through inverters. If there is an even number of inverters on the $w(v)$-v path we directly attach v to $w(v)$. Else, we attach v to the (only) inverter remaining in that net.

This concludes the preprocessing. An illustration of all the taken steps can be seen in Figure 5.2.

5.3.2 Subcone collection

We start by collecting what we call subcones and will later merge them whenever we figure out that they should be part of the same instance.

Definition 5.3.2. Let C be a circuit. A **subcircuit** C' of C is a circuit such that the underlying graph of C' is a subgraph of C. A **subcone** with root r in C is a subcircuit T of C such that for every $v \in \mathcal{G}(T)$, there is a v-r-path contained in T. The set $\mathcal{I}(T) := \delta^-(T) \setminus T$ is called the **set of inputs** of T.

Observation 5.3.3. Every subcone contains its root r and is connected.

We study suitable subcones of our circuit and aim to find the set of maximum AND and XOR subcones.

Definition 5.3.4. Let G be the graph underlying a circuit C. A subcone of G is called an **AND subcone** if every vertex in the subcone is an AND gate. Analogously, a subcone is called an **XOR subcone** if it consists only of XOR gates.

The collection of maximum subcones can be done similarly to Algorithm 5.1. We just remove the condition that a vertex should have exactly one successor when being added to the subcone. Thus, we go through the circuit in inverse topological ordering. For every 2-ary vertex that is not in a previously collected subcone we create a new subcone with this vertex as root. For that we consider all the predecessors. If a predecessor has the same gate type as the root, we add it as a gate to the circuit and consider its predecessors recursively. Else, the predecessor becomes an input. This results in Algorithm 5.2. We remark that the collected maximum subcones are not necessarily disjoint.

Note that the subcone collection applied to Figure 5.2(d) yields one XOR subcone and two AND subcones. One of the AND subcones consist only of the gate succeeding t_4 and t_5 while the other consists of all other AND gates.

5.3.3 Merging related subcones to instances

To derive our instances, the maximum subcones need to be merged if they have potential for common optimization. For this, we study the sets of inputs.

If two collected subcones share two inputs, we want them to be in the same instance. If one subcone takes a vertex v as an input and another subcone takes the inverted version of v as an input, then their input sets do not overlap at this input. The reason is that the input of the second cone is the inverter following v and not v itself. Due to our preprocessing of moving inverters through XOR gates, this cannot happen for XOR components though. For AND subcones however, we do not want them to intersect at this input since no vertex computing $v \wedge w$ for any w can be used by the final output of the second subcone. Thus, the behavior is as desired.

To compute the instances, we generate an undirected auxiliary graph H containing a vertex for every subcone and an edge if they share at least two inputs. In this auxiliary graph, we compute the connected components and define every component to be one instance. To derive the collection of trees in the form used in the definition

Algorithm 5.2: Subcone collection

Input: a circuit $C = (\mathcal{V}, \mathcal{E})$ and an operator $\circ \in \{\textsc{And}, \textsc{Xor}\}$.
Output: a family $\mathcal{F}$ of (not necessarily disjoint) $\circ$ subcones.

$\mathcal{F} = \emptyset$
for $w \in \mathcal{V}$ *in inverse topological order* **do**
  if $w \in T$ *for some* $T \in \mathcal{F}$ *or gate type* $\phi(w) \neq \circ$ **then**
    continue
  $S \leftarrow \{w\}$
  $T \leftarrow \emptyset$
  while S *is not empty* **do**
    Remove a vertex $v \in S$ from S.
    if *gate type* $\phi(v) = \circ$ **then**
      Add v to T.
      Add the predecessors of v to S.
  Add T to $\mathcal{F}$.
return $\mathcal{F}$

of ATSO, one can simply run over all subcones of the connected component. For every vertex v in the subcone that has successors outside the connected component, we add all the inputs that $\mathcal{I}_v$ depends on as a set to our instance. Thereby, we ensure that after computing a new solution, we still compute every result used by another part of the chip. This procedure is captured in Algorithm 5.3.

Algorithm 5.3: ATSO instance collection

Input: a circuit C and an operator $\circ \in \{\textsc{And}, \textsc{Xor}\}$.
Output: a collection $\mathcal{S}$ of ATSO instances.

$\mathcal{F} = \textsc{Subcone Collection}(G, \circ)$.
Let $H = (\mathcal{F}, \mathcal{D})$ where $\mathcal{D} \coloneqq \big\{\{F, F'\} \mid |\mathcal{I}(F) \cap \mathcal{I}(F')| \geq 2\big\}$.
Compute the set of connected components $\mathcal{C}(H)$ of H.
for *every component* $X \in \mathcal{C}(H)$ **do**
  $\mathcal{T} = \emptyset$
  for *every subcone* $Y \in V(X)$ **do**
    for $v \in Y$ *for which* $\delta_C^+(v) \setminus \left(\cup_{Y' \in V(X)} V(Y')\right) \neq \emptyset$ **do**
      $T = \text{cone}_Y(v) \cap \mathcal{I}(Y)$
      $\mathcal{T} = \mathcal{T} \cup \{T\}$
  $\mathcal{S} = \mathcal{S} \cup \{\mathcal{T}\}$
return $\mathcal{S}$

5.4 Application of the integer linear program

Once the instances are collected we can use the LP formulation described in Section 4.6. As the majority of the instances is not very large, we use an integer linear programming formulation. This cannot be solved in polynomial time but integer linear program (ILP) solvers are still often fast in practice. We use the

(ATSO LP relaxation) and transform it into an integer linear program by requiring that every variable is integral.

$$\begin{aligned} \min \quad & \sum_{S \in \mathcal{S}_2} x_S \\ s.t. \quad & \sum_{\{S', S\setminus S'\} : \emptyset \subsetneq S' \subsetneq S} \min\{x_{S'}, x_{S\setminus S'}\} \;\geq\; x_S \qquad \forall S \in \mathcal{S}_2 \\ & x_T \;=\; 1 \qquad \forall T \in \mathcal{T} \\ & x_S \;\in\; \{0,1\} \quad \forall S \in \mathcal{S} \;. \end{aligned}$$

(ATSO ILP)

Our objective is to reduce the number of 2-ary gates in our representation. In the future, it might be worth to try to include different objectives including delay and possibly even netlength. We will give more details on such extensions in Section 5.7.

5.4.1 ILP solvers

There are different ILP solvers available that can be used to solve this ILP. We have tried to use the commercial general-purpose ILP solvers CPLEX and Gurobi. The results were very similar and also the running times were very similar, possibly CPLEX was slightly faster. For the tests shown in Section 5.6 we have thus used CPLEX.

5.4.2 ILP speedup techniques

The ILP contains an exponential number of variables and constraints in k. For large values of k this leads to very large running times. To avoid this, we try to reduce the instance size before solving the ILP to speed up the solver.

The ILP size can be reduced by merging inputs that occur in exactly the same set of trees. This is shown in the following lemma.

Lemma 5.4.1. *Let $\mathcal{T}$ be an ATSO instance. Assume that there exist $i, j \in \{1, \ldots, n\}$ such that for every $T \in \mathcal{T}$ either $T \cap \{i, j\} = \emptyset$ or $T \cap \{i, j\} = \{i, j\}$. Then there always exists an optimum solution in which one vertex computes $i \wedge j$ and this vertex is the only successor of i and the only successor of j.*

Proof. Assume C is an optimum solution for which this is not true. Introduce one additional vertex v computing $x_i \wedge x_j$ and whenever x_i is the predecessor of a vertex, use v instead. Since i and j are contained in the same set of trees, this does not change the functions computed at the outputs. Moreover, whenever x_j is the input of a vertex w, remove this vertex and use the other predecessor of w as an input for the successors of w. Since variable x_i is added to the computed function at some point, too, this also does not change the computed function due to our modification of taking v instead of x_i. Thus, after these modifications we still have a valid solution for our instance.

We have added one gate and have removed at least one gate since there has been at least one vertex whose input x_j was. Hence, our solution did not become more expensive and is still optimum. □

Before giving the ILP to a solver, we can sometimes significantly reduce the size by substituting such two inputs x_i, x_j as mentioned in Lemma 5.4.1 with a single new input $x_{\bar{i}}$. After solving the resulting pruned ILP, we then add one additional gate to compute $x_{\bar{i}} = x_i \wedge x_j$.

Figure 5.3 shows the ratios of how much the sizes can be decreased by the application of Lemma 5.4.1. It shows that the majority of the instances can be reduced to smaller k, and most of them can even be reduced significantly.

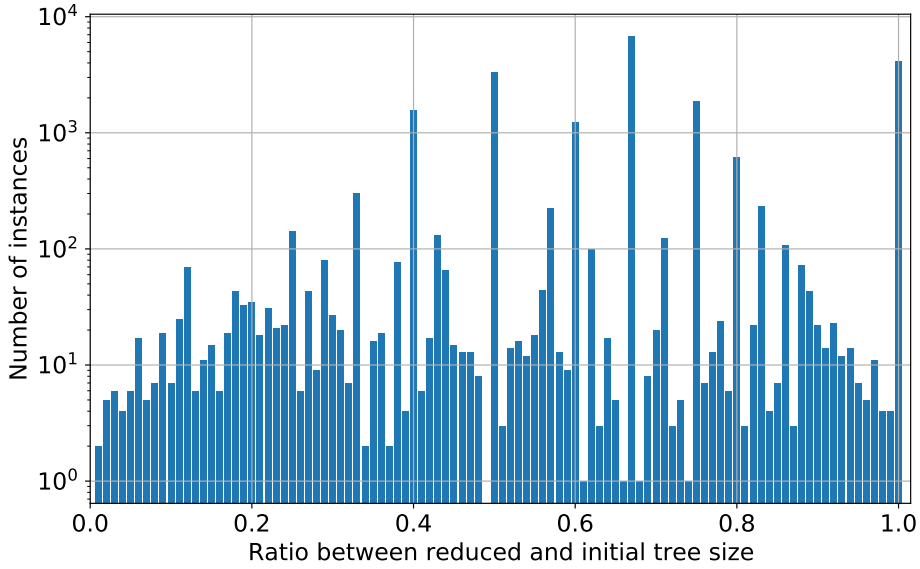
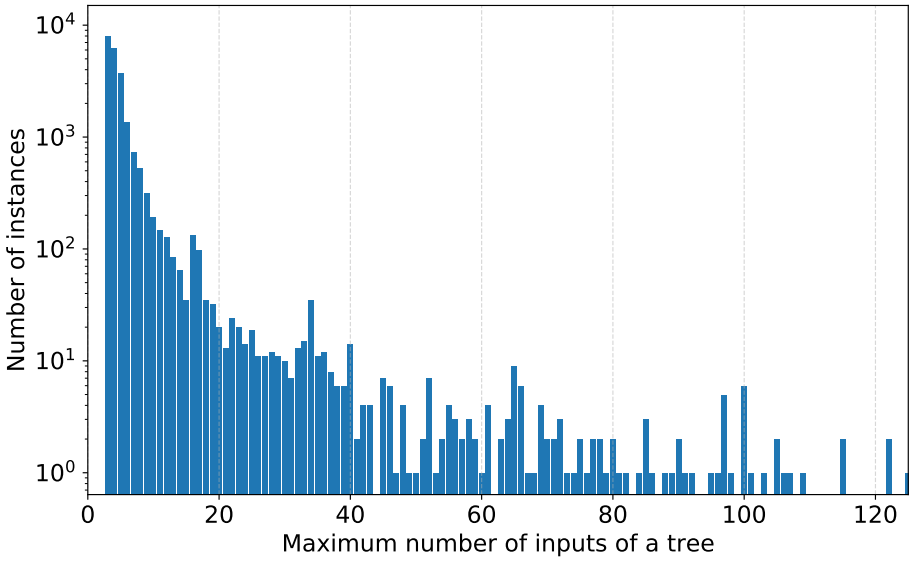


Figure 5.3: Relative ratios of the reduced tree size compared to the initial tree size. This statistic has been taken over all 14 chips and 19 benchmarks described in Section 5.6.

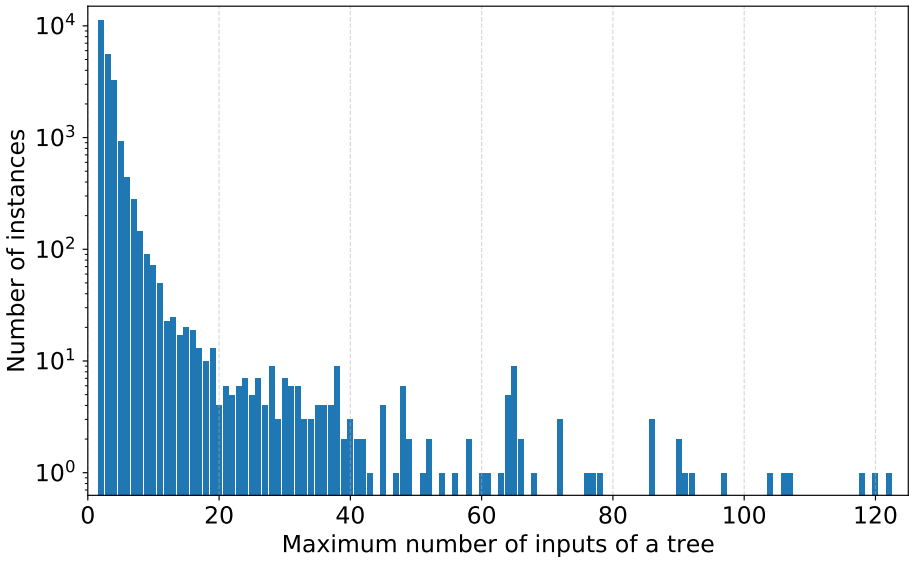
To also see how large this number k is in absolute terms, we consider Figure 5.4. It shows a summary over the different maximum tree sizes and their frequencies for instances as they are found on the chips and benchmarks. Figure 5.4(a) shows these numbers before the application of Lemma 5.4.1 and Figure 5.4(b) afterward. One can see that most of the instances have at most 40 inputs, a lot of them even have a maximum of 10 inputs. Moreover, this plot also shows that the tree sizes are decreased significantly. Note that both plots have a logarithmic y-scale.

To further improve the speed with which a good solution to our problem can be found, we give an upper bound on it to the ILP solver. Our upper bound is the solution value of the implementation of the instance on the chip. Note that when applying Lemma 5.4.1, we cannot give it the full solution but only its value. Our tests however showed that it is much quicker to have a smaller ILP with an upper bound on the optimum solution than the non-pruned ILP with a full initial solution.

Note also that we usually use the restriction of Section 4.1.3 when solving the integer linear program, that is we only allow for every input to have at most one path to every output. As already observed in Section 4.1.3, this can lead to worse solutions. Thus, it might be possible that the value of the ILP is larger than the value of our initial solution which not necessarily follows this restriction. However, allowing for arbitrary many paths would make the ILP different for Xor and And trees, i.e., one would need two implementations. What is more important though, one would additionally increase the number of variables and constraints a lot. This is due to the fact that we would need the first constraint of the (ATSO ILP) not only for partitions but for general covers instead. Taking these two reasons into



(a) Initial tree sizes



(b) Tree sizes after application of Lemma 5.4.1.

Figure 5.4: Plots of the tree sizes before and after the application of Lemma 5.4.1. These plots were taken over 14 chips and 19 benchmark instances. Initially, there are 22 instances that have been excluded from the plot in Figure 5.4(a) since they contain more than 125 inputs in a single tree. After application of the size reduction, there are only 8 of them remaining that have been excluded from Figure 5.4(b).

consideration, we decided to implement the restricted ILP.

The tests we made justify this approach. Among all the chips and benchmark instances we have used to test our approach, there were only very few instances on which the initial solution did not satisfy the mentioned restriction. Even more importantly, on none of these few instances the optimum solution under the restriction was worse than the initial solution without it. Thus, even if this is a restriction as noted in Figure 4.2, it does not have practical relevance.

Last, we set some additional bounds to ensure that no single instance leads to huge running times. We set the maximum running time of the ILP solver to 100 seconds. When running into this limit, we just use the best solution that was computed at this point. Moreover, we restrict ourselves to solving instances where a single tree has at most 15 inputs. Else, already the constraint generation can take really long, which is not included in the runtime limit. For solving the ILP we use 32 threads.

5.5 Inserting the ILP solution onto the chip

Once an ILP solution is computed, this solution needs to be put back to the chip and undergo a set of improvements. The first step is to compute a technology mapping on which we elaborate in Section 5.5.1. Since the ILP does not choose a good implementation for the subtrees contained in our instances, we also run the usual tree restructuring as described in Section 5.5.2. After that, we apply a buffering routine to get reasonable slacks (see Section 5.5.3). Moreover, we need a reasonable placement for the new gates as our instances can have a completely different structure afterward. We describe the different placement steps in Section 5.5.4.

5.5.1 Technology mapping

To insert the instances back onto the chip, we use the technology mapping tool contained in the BonnTools. This tool has not been the focus of this work, but we describe it to explain some issues we see with it. Technology mapping in general is the step of mapping given Boolean functions onto gates contained in the library. In our case we have a representation of the circuit using 2-ary gates and inverters only. The technology mapping restructures this circuit making local improvements. The main algorithm framework is due to Elbert [Elb17] and has been further developed by Hermann [Her20] and Armbruster [Arm21]. We describe it in the following and start by introducing a few concepts.

Definition 5.5.1. Let v be a vertex in a circuit C.

(i) The **cone** $\text{cone}(v)$ of a vertex $v \in \mathcal{V}$ is the subgraph induced by all vertices $w \in \mathcal{V}$ for which there exists a w-v-path in C. We say that v depends on w if $w \in \text{cone}(v)$.

(ii) A ***v*-cut** X is a subset $X \subseteq \mathcal{E}[\text{cone}(v)]$ of the edges of the cone satisfying the following properties.

  (a) For every input $i \in \text{cone}(v)$, every i-v-path contains exactly one edge of X.

  (b) The connected component of v after removing X from $\text{cone}(v)$ does not contain a vertex with fanout at least two apart from v.

We start our technology mapping algorithm by computing all possible matches in the following sense.

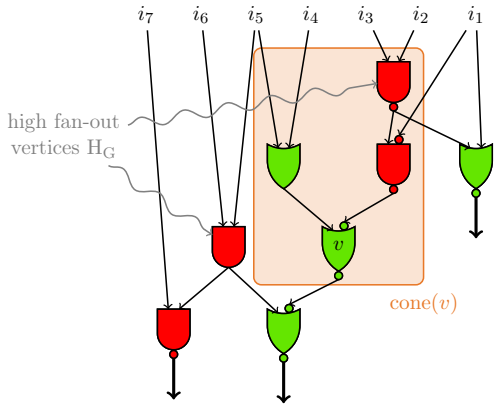
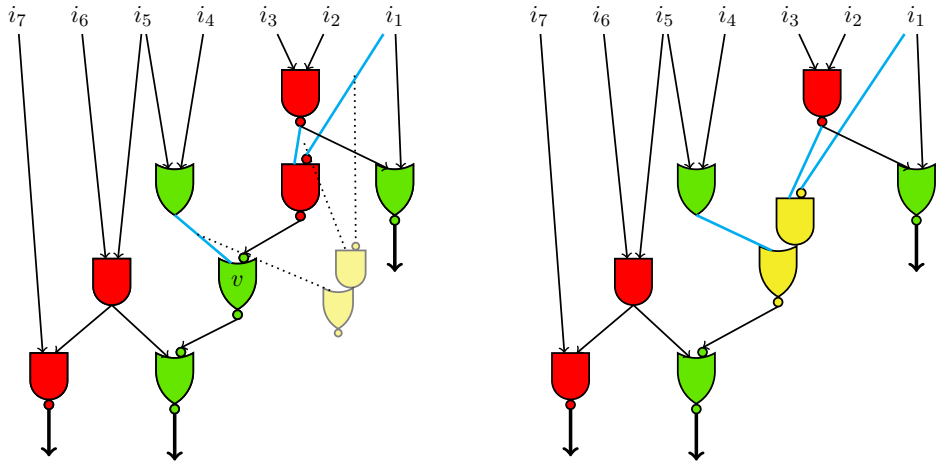


Figure 5.5: Example of a technology mapping instance with a chosen vertex v. The cone of v is shown in orange.

Definition 5.5.2. Let C be a circuit over library $\mathcal{L}$. A **match** m on $v \in \mathcal{G}(C)$ is a subcircuit M consisting of one gate $g \in \mathcal{L}$ and possibly some inverters together with a v-cut X_m satisfying the following property. When replacing the connected component of v in $\text{cone}(v) \setminus X$ by M the resulting circuit is equivalent to C.

For inputs $i \in \mathcal{I}(C)$ there shall exist one trivial match for i.



(a) A v-cut shown in blue and a possible match shown in yellow.

(b) Equivalent circuit in which the match on the left got realized.

Figure 5.6: Example of a match for the circuit shown in Figure 5.5.

Matches for different nodes are then combined to form candidates.

Definition 5.5.3. A **candidate** can_v for a vertex v is a match m for v together with candidates can_i for all input pins i of the match m, i.e., for all tails of edges in X_m.

Such a candidate naturally induces candidates for the predecessors of the match and therefore induces a whole circuit to compute the boolean function of the match. Our goal is to compute a candidate for every output, and our algorithm ensures

that there does not exist a vertex on which two candidates imply different matches. The algorithm is implemented by a dynamic program which goes over all vertices in topological ordering and computes all possible candidates. To ensure that two candidates imply two different matches on a vertex, we restrict to one candidate whenever a vertex has more than one successor. The described algorithm is summarized as Algorithm 5.4.

Algorithm 5.4: General technology mapping algorithm

Input: a circuit C with arrival times $\mathrm{AT}(i)$ for all inputs $i \in \mathcal{I}(C)$ and required arrival times $\mathrm{RAT}(o)$ for all outputs $o \in \mathcal{O}(C)$.

Output: an equivalent circuit possibly satisfying the required arrival times.

```
1 Compute all matches.
2 for each vertex v in topological order do
3     Compute all possible candidates from all found matches.
4     Delete all non-dominated candidates for v.
5     if v has more than one successor then
6         Restrict to the best candidate.
7     if v is an output then
8         Restrict to the best candidate.
```

The possible objectives of the algorithm are timing, size and netlength. There are two places where one can tune the algorithm, i.e., where the algorithm needs to choose some candidates according to the chosen objectives. On the one hand, there is a candidate fixed for each output as follows. If there exists a candidate satisfying the required arrival time, choose the candidate with smallest netlength, size and delay among all such candidates. Else, choose the fastest candidate. The exact tradeoff between size and netlength can be specified. On the other hand, a candidate is fixed at every vertex with at least two successors. This choice is done by minimizing a linear combination of the three optimization directions.

Since tuning this technology mapping only on minimum size leads to very bad timing values, we stick to the default linear combination.

5.5.2 Restructuring the contained trees

Every ATSO instance can be decomposed into trees. Such a single tree is not properly restructured by the ILP: the current version of the ILP only optimizes on size and for a single tree as presented in Section 5.2.2, there is nothing to optimize. Thus, every single-output subtree is chosen basically randomly. Hence, it makes sense to apply the routine of Section 5.2.2 to every subtree to improve the timing and netlength properties. We do this after the technology mapping.

5.5.3 Buffering

We initially work on a buffered instance even though we use a delay model without wire delays because our delay model includes bifurcation delays. Thus, nets with many sinks are buffered to avoid hugh such delays. To make timing results more comparable, we apply the same buffering routine on the derived instances afterward. For this, we first rip out all but one inverter for every net. This can be done maintaining logical correctness. Afterward, we buffer all the nets. Note that this buffering is non-deterministic which makes our results non-deterministic, too, unfortunately.

5.5.4 Placement of AND components

To place the components, we first place each gate at the median of its inputs and a position that is best regarding the required arrival time. Such a position can be computed as described by Li et al. [Li+10]. We first compute the best position with respect to required arrival time in inverse topological ordering and then compute the median with respect to both x and y direction. This choice yields a good position with respect to timing but not necessarily a good position when considering netlength. Thus, we encountered sometimes a significant increase in the netlength on some of the instances. To avoid this, we apply a state-of-the-art flat placer after the restructuring of the contained trees.

5.6 Experimental results

In this section we show some tests on practical instances. We start by describing our testbeds and metrics. Our testbed consists of two sets of instances.

First, we use a set of instances obtained from the EPFL benchmarks on logic restructuring by Amarú, Gaillardon, and De Micheli [AGD15]. They are mapped to the **NanGate45 Open Cell** library, which is an open library providing 45nm cells. We have used this alternative library instead of the **ASAP7 Cell** library because the latter was providing very inconsistent sizes for different arities of AND gates. Since small AND gates were only available in very large sizes and an AND5 gate for example was only available in a very small size, standard AND2 and AND5 gates were about the same size. This does not at all reflect gate sizes in a practical modern library and led to strange results. Hence, we have chosen to stick with a more realistic cell library instead.

Second, we perform tests on industrial chip instances provided by IBM. Each of the chips is in 5nm technology. Half of the chips, that is chips 1, 2, 7, 8, 9, 10 and 13, are server chips, the other chips are ASICS. Due to non-disclosure agreements we cannot give further details on the chips. We enumerated them consistently throughout this thesis sorted by their number of gates after ripping out inverters (see line 9 of Algorithm 5.5).

These described sets of instances are the same as the ones used to generate Figure 5.3 and Figure 5.4.

All our tests have been executed on a Rocky Linux 8 4.18 machine with an AMD EPYC 9684X processor.

To improve comparability between different runs we have saved an initial description of each instance that we call snapshot. Every snapshot contains a list of cells, nets, placement data and much more information at a certain step in the design flow. Our chosen state is after a first placement and an additional buffering based on a delay model without wire delay. By at first loading this state, we ensure that we are independent of the nondeterminism of the initial technology mapping on the EPFL benchmarks and this initial buffering. Note that due to the buffering running at the end of our flow, our results are still partially non-deterministic. Moreover, there might be small changes due to the limit on the running time of the ILP solver.

Based on the previous sections, we summarize our algorithm in Algorithm 5.5.

5.6.1 Practical examples

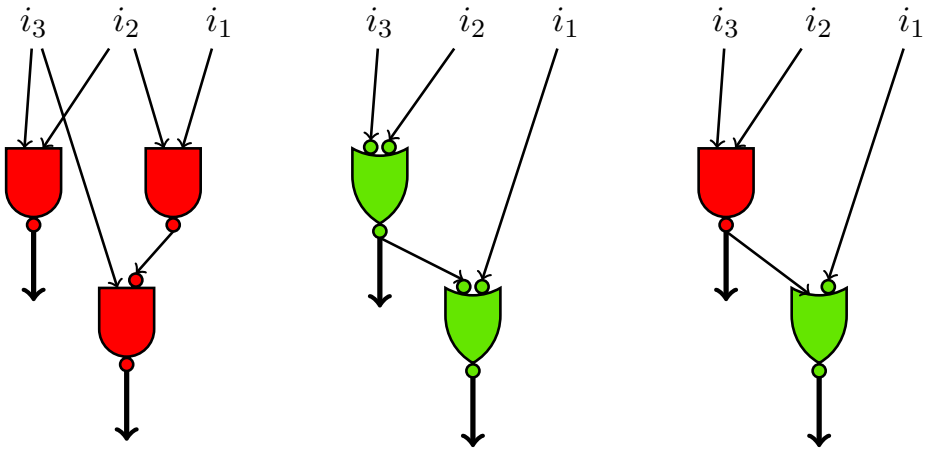
In the following we show two examples on which our algorithm worked as expected and was able to reduce the number of gates. This at the same time gives some intuition on why our algorithm is useful. All instances are taken from the EPFL

Algorithm 5.5: Summary of the flow used for the tests in Section 5.6.4.

```
1  Load snapshot.
2  Activate timing model without wire delay.
3  Collect ATSO instances (see Section 5.3).
4  for each instance with maximum tree size at most 15 do
5      Restructure instance using the ILP solver (see Section 5.4).
6      if ILP yields an improvement then
7          Map them to the chip by technology mapping (see Section 5.5.1).
8          Restructure the trees by And resp. Xor tree restructuring (see
           Section 5.5.2).
9  Rip out all but at most inverters per net.
10 Buffer all nets (see Section 5.5.3).
11 Improve placement (see Section 5.5.4).
```

benchmark instance *multiplier*.

For both examples we show three plots. First, we show a plot of the initial instance. Then we show a plot after the restructuring by the ILP solver. Note that since not every chip has available And cells, we directly apply a normalization step and implement the instance based on Nor gates instead. The third plot shows the instance after technology mapping.



(a) Initial instance before our routine.

(b) Restructured instance after ILP application.

(c) Instance after technology mapping.

Figure 5.7: Restructuring of an instance with three gates. Note that the result computes the inverted function of the main output.

Note that as one can see for example in Figure 5.7, we do not plot inversions after the output. In this case, the solution returned by the ILP solver and by technology mapping compute the inverted function compared to the initial instance. When the technology mapping compares the cost of different solutions, it takes into account whether an inverter will be needed. Sometimes it might be possible to save one inverter by computing the inverted function, sometimes the number of inverters is equal, and sometimes an inverted candidate implies an additional inverter.

Figure 5.7 shows an easy example where our routine noted that one gate can be saved by restructuring the trees for one of the outputs. There are many equivalent instances distributed over the different chips and benchmarks.

Figure 5.8 shows a more complicated instance. It has ten gates initially and can be reduced to eight gates, six of which are outputs. This instance is a bit unusual because it is considered to have 8 inputs. When computing the instances, we decompose every gate into AND and inverter gates, and between gate g and its successor h one inverter remains. This inverter is not part of the instance. It is at the same time a successor of gate g and an input of gate h. Such instances are consistent with our definition of an instance. Nevertheless, it requires a careful implementation while reinserting the instance onto the chip because this inverter does not exist in the initial instance. There are many such instances, in which inverters that do not even exist in the initial instance then form both a successor of a gate and an input of another gate. Some of these instance also contain many such inverters.

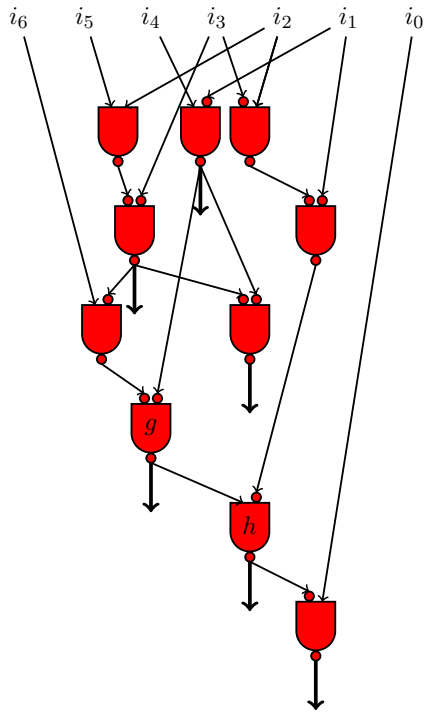
Note that the depth of this instance increases during our procedure. This can however easily be fixed by restructuring the tree above output h and implementing it as in the original instance. This is an option that is considered in line 8 of Algorithm 5.5. Whether this solution is really restructured in that fashion in order to decrease the depth depends on the arrival times and the netlength of this instance.

5.6.2 Reduction of cloning

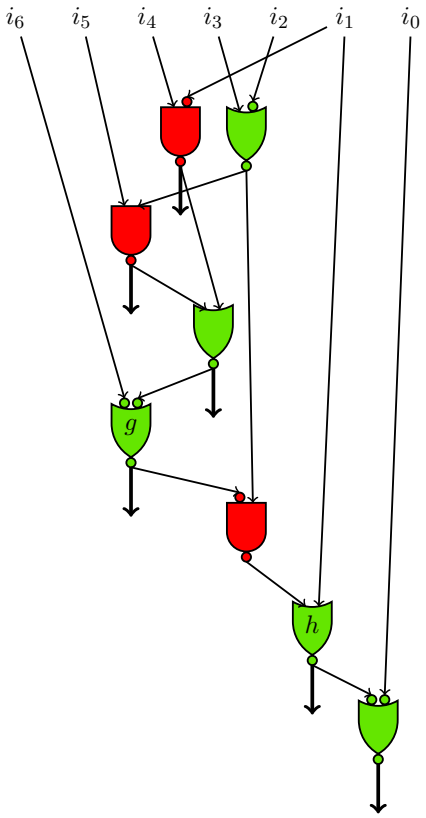
During this algorithm we also discover what we call **cloned gates**, i.e., sets of gates that compute the same or just the inverted function of one another. Mostly, they occur in pairs but sometimes, more than two gates compute the same function. In these cases the algorithm saves one (ore sometimes several) gates by reusing the result that is computed for all the sinks. There are cases in which one might want to have two gates computing equivalent functions though. The most important one applies for timing critical nets with many sinks. Bifurcations in nets cost some delay. Thus, one wants to avoid having many sinks in the same net if this net is timing critical. One way to do so is to compute the predecessor result multiple times and then distribute the sinks needing this signal onto the different copies. This can of course also have some advantages as the copy of the predecessor can already be placed in the direction of the succeeding sinks. Nevertheless, in most of the instances, especially for timing uncritical ones or vertices with few successors, this is not the case. In these cases, it makes sense to combine the equivalent gates into one and save some space and power. We allowed our routine to be restricted to optimizing these instances only to see how often this occurs. The result can be seen in Table 5.1. We show the total number of gates after ripping out all but one inverter per net, the number of gates that we remove since they are cloned and the total number of 2-ary gates that the ILP could remove.

One can observe that on the chips up to 0.6% of the cells are cloned cells that one can get rid of if desired. On most chips, these numbers are small compared to the total number of 2-ary gates we remove while solving the ILP. On chip 14 though, about 40% of the removed cells are cloned cells.

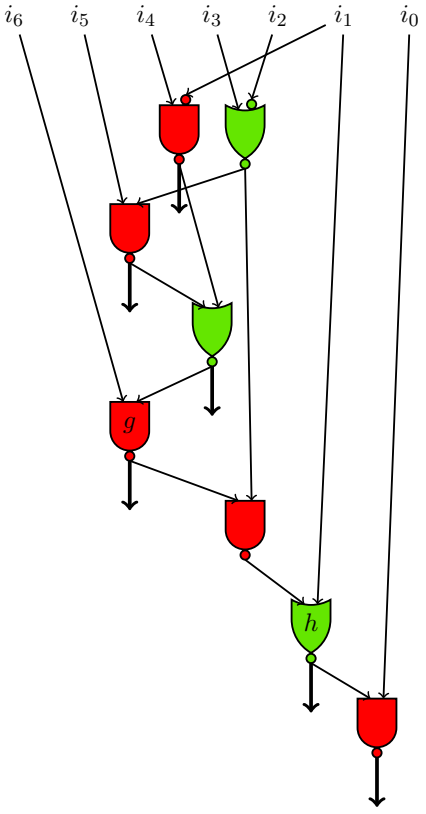
We remark that we only show results on the chips in Table 5.1 and not on the benchmarks since there is no cloning on the benchmarks due to their different initial technology mapping.



(a) Initial instance containing 6 outputs and 10 gates.



(b) Restructured instance after ILP application with 8 gates.



(c) Instance after technology mapping.

Figure 5.8: Instance with a gate g which is an internal input.

Chip	#Gates	#Cloned gates	#Removed gates
Chip 1	6221	0	134
Chip 2	15 134	97	391
Chip 3	17 814	2	100
Chip 4	41 362	5	125
Chip 5	48 378	12	493
Chip 6	64 904	102	2193
Chip 7	68 138	78	1178
Chip 8	80 086	77	2811
Chip 9	79 523	86	1348
Chip 10	162 572	108	8120
Chip 11	140 157	240	1208
Chip 12	282 498	155	2810
Chip 13	427 449	198	3150
Chip 14	1 019 499	2169	5408

Table 5.1: This table shows the total number of gates on all of the chip, the number of gates that are cloned from another gate and the total number of 2-ary gates that we remove while solving the ILP. We remove all the cloned gates in our routine.

5.6.3 Reduction of 2-ary gates due to the ILP

The main step of our new flow is the reduction of 2-ary gates that can be achieved by the ILP solver. Most of the instances are solved optimally so this reduction cannot be improved much further. We thus show some quantative results on how much one can gain in terms of 2-ary gates on chips whose logic has been optimized with state-of-the-art tools in advance. We visualize this improvement using different plots and tables.

Figure 5.9 shows the relative reduction of 2-ary gates while solving the ILP over all the instances. A ratio of 0.5 thus means that our initial instance needed twice as many gates as the solution found by the ILP. We omit instances on which the ILP could not derive any improvement in this plot. These are a total of 11988 instances. The plot shows that one can reduce the instance size on average by 20% on instances on which the size can be reduced at all (13887 instances). Hence, on more than half of the instances the number of 2-ary gates is not optimum. Moreover, there are several instances, on which the number of 2-ary gates can be reduced significantly, sometimes even by more than 50%.

Figure 5.10 shows a distribution of the different reductions by the initial number of gates in the instance over instances with at most 250 gates. There are some larger instances that have been omitted in the plot for better readability.

To see how our testbed behaved under this routine, we consider Table 5.2. This table shows both the results for our set of chips and for the EPFL benchmark instances. One can see that there is a significant decrease in the number of 2-ary gates. Instances on real world chips seem to allow for a better improvement than the benchmarks, which can of course well be related to the different initial technology mapping routines. Moreover, the reduction of cloning might also play a role in this discovery. However, on chip 10, which is the chip with the largest reduction by 22%, only 1.3% of the reduced gates was removed due to cloning.

On some of the benchmarks there are only few different instances. This might

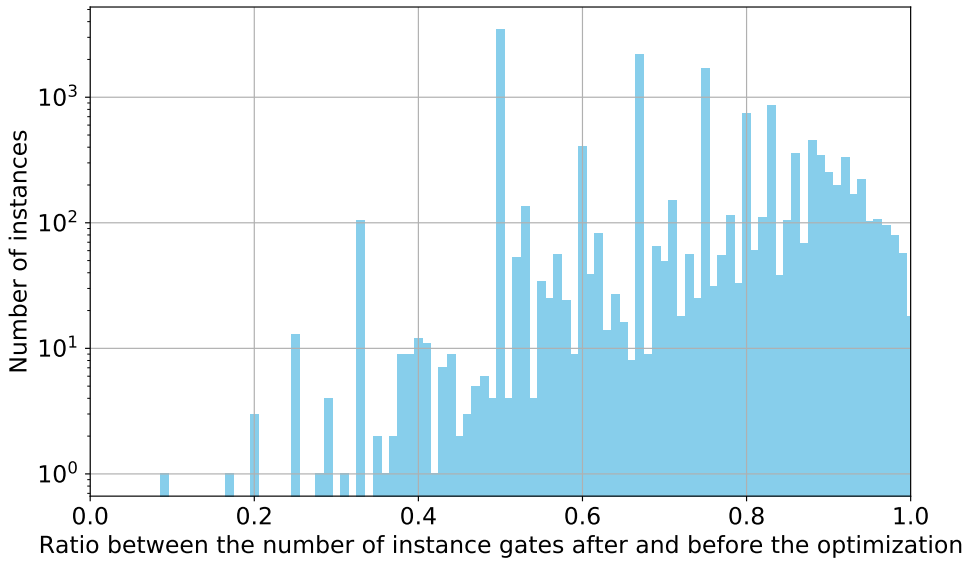


Figure 5.9: Relative ratios of the number of 2-ary gates after and before the ILP application. This statistic has been taken over all chips and EPFL instances. We omit the 11988 instances that could not be improved using our restructuring method.

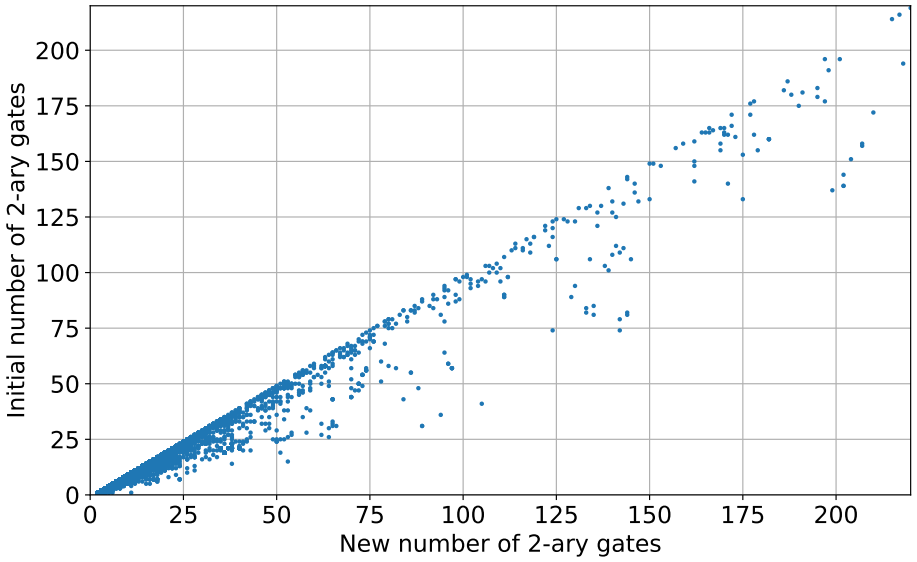


Figure 5.10: Reduction of the number of 2-ary gates in our internal model. Every dot implies a reduction of the initial number modeled by its x-coordinate to the reduced number modeled by its y-coordinate on at least one instance.

Chip	#Gates in instances	#Gates after optimization		#Instances
adder	495	454	−8.28%	46
arbiter	5083	5083	0.00%	2
bar	1680	1680	0.00%	2
cavlc	348	345	−0.86%	6
ctrl	57	57	0.00%	2
dec	304	304	0.00%	1
hyp	120 789	118 975	−1.50%	5809
i2c	545	544	−0.18%	8
int2float	125	122	−2.40%	2
log2	18 788	18 405	−2.04%	801
max	2120	1978	−6.70%	306
mem_ctrl	15 720	15 642	−0.50%	443
multiplier	10 736	10 355	−3.55%	968
priority	357	357	0.00%	1
router	192	184	−4.17%	4
sin	4282	4230	−1.21%	110
sqrt	22 692	20 718	−8.70%	1172
square	5128	5075	−1.03%	447
voter	2581	2573	−0.31%	562
Chip 1	2190	2056	−6.12%	41
Chip 2	2838	2447	−13.78%	231
Chip 3	2403	2303	−4.16%	76
Chip 4	4783	4658	−2.61%	160
Chip 5	9871	9378	−4.99%	544
Chip 6	14 262	12 069	−15.38%	746
Chip 7	13 376	12 198	−8.81%	654
Chip 8	19 580	16 769	−14.36%	441
Chip 9	19 503	18 155	−6.91%	749
Chip 10	36 511	28 391	−22.24%	1528
Chip 11	14 207	12 999	−8.50%	701
Chip 12	117 517	114 707	−2.39%	1418
Chip 13	65 901	62 751	−4.78%	2051
Chip 14	175 170	169 762	−3.09%	4772

Table 5.2: This table shows the reduction of the number of 2-ary gates in our instances. The first column shows the total number of 2-ary gates that are part of our instances. The second column shows their number after reducing them using the ILP solver. The last column shows the total number of instances over which these gates have been distributed.

be influenced by the number of gates that exist on these chips. Most of the EPFL benchmarks contain fewer gates than the average chip, as one can also see in Section 5.6.4.

5.6.4 Overall results

We now show the overall results when running Algorithm 5.5. We compare two runs. The first run called *tm* is Algorithm 5.5 in which line 5 is skipped. In this run we still solve the ILP and if we find a better solution, we apply technology mapping and the tree optimization to the initial solution. This implies that we apply technology mapping and the tree optimization (i.e., line 7 and line 8) to an instance if and only if Algorithm 5.5 would do so, too. Note that for a fair comparison, we also decompose our circuit into 2-ary gates before technology mapping as described in Section 5.3.1. The second run called *atso* is the run given in Algorithm 5.5. The only difference between the two runs is therefore whether we use the ILP solution that we have computed or not.

We evaluate the runs with respect to the different metrics that we explain in the following. All line numbers refer to Algorithm 5.5.

Total area:	Area of all gates and repeaters measured after ripping out inverters in line 9.
#Gates before RipOut:	Number of gates before ripping out inverters in line 9.
#Gates after RipOut:	Number of gates after ripping out inverters in line 9.
#Gates w\o inv:	Number of gates without repeaters, i.e, without inverters and buffers after line 9.
Netlength:	Steiner netlength after placement in line 11 in meter.
Worst Slack:	Minimum slack encountered at any gate in picoseconds. This metric is measured after the final placement in line 11 in a delay mode without wire delay.
ILP runtime:	Total runtime of the ILP solver on all instances.
Runtime:	Total runtime of the main algorithm, including instance collection, ILP solving, technology mapping and And and Xor tree optimization. This corresponds to lines 3 to 8.

The results of our tests can be seen in Table 5.3 and Table 5.4. Due to the large number of test instances, we show the results for the benchmarks and for our set of chips separately.

The results on the EPFL benchmarks in Table 5.3 look very good overall. The total area decreases on every chip but on *multiplier* if we found at least one instance that we were able to optimize (else it stays the same). On *router*, it even decreases by 23%. On average, we can see a decrease of about 1.6% among all gates on all benchmarks. This area decrease goes along with a large reduction of the number of non-repeater gates, and a smaller but still significant reduction in the overall number of gates. The number of non-repeater gates decreases by up to 14.3% and the number of gates including inverters decreases by up to 16.5% after RipOut. Both of these values are achieved on *router*. On most of the instance, the table still implies that the number inverters decreases due to our routine: the difference between the number of gates before RipOut is rather larger than the difference between the number of gates without inverters. Thus, our flow does not only decrease the number of 2-ary

Chip	Run	Total Area		#Gates before RipOut		#Gates after RipOut		#Gates w\o inv		Netlength		Worst Slack		ILP runtime	Runtime
adder	tm	53 655		2425		2220		1297		0.029		−0.7		00:00:00	00:00:01
	atso	51 979	−3.12 %	2354	−2.93 %	2187	−1.49 %	1243	−4.16 %	0.028	−3.45 %	−1.5	−0.9	00:00:00	00:00:01
arbiter	tm	229 455		9811		8846		5623		0.158		−0.3		00:00:01	00:00:05
	atso	229 455	0.00 %	9811	0.00 %	8846	0.00 %	5623	0.00 %	0.157	−0.63 %	−0.3	0.0	00:00:01	00:00:05
bar	tm	133 192		5315		4903		3600		0.054		−0.3		00:00:00	00:00:01
	atso	133 192	0.00 %	5315	0.00 %	4903	0.00 %	3600	0.00 %	0.054	0.00 %	−0.3	0.0	00:00:00	00:00:01
cavlc	tm	30 282		1273		1104		647		0.009		−0.2		00:00:00	00:00:01
	atso	28 952	−4.39 %	1221	−4.08 %	1093	−1.00 %	611	−5.56 %	0.007	−22.22 %	−0.3	0.0	00:00:00	00:00:01
ctrl	tm	3492		157		140		83		0.001		−0.1		00:00:00	00:00:00
	atso	3492	0.00 %	157	0.00 %	140	0.00 %	83	0.00 %	0.001	0.00 %	−0.1	0.0	00:00:00	00:00:00
dec	tm	15 803		655		616		304		0.004		−0.1		00:00:01	00:00:01
	atso	15 803	0.00 %	655	0.00 %	616	0.00 %	304	0.00 %	0.004	0.00 %	−0.1	0.0	00:00:01	00:00:01
hyp	tm	10 621 080		414 910		373 151		251 984		3.249		−215.9		00:03:33	00:38:33
	atso	10 506 654	−1.08 %	411 837	−0.74 %	370 485	−0.71 %	249 166	−1.12 %	3.173	−2.34 %	−239.9	−24.0	00:03:34	01:03:07
i2c	tm	47 995		2065		1806		1114		0.027		−0.2		00:00:00	00:00:00
	atso	47 955	−0.08 %	2066	0.05 %	1809	0.17 %	1112	−0.18 %	0.026	−3.70 %	−0.2	0.0	00:00:00	00:00:00
int2float	tm	10 541		432		394		217		0.002		−0.2		00:00:00	00:00:00
	atso	9546	−9.45 %	407	−5.79 %	375	−4.82 %	200	−7.83 %	0.002	0.00 %	−0.2	0.0	00:00:00	00:00:00
log2	tm	1 376 460		58 016		51 838		32 930		0.564		−4.9		00:00:10	00:00:54
	atso	1 356 072	−1.48 %	57 390	−1.08 %	51 358	−0.93 %	32 370	−1.70 %	0.547	−3.01 %	−5.4	−0.5	00:00:10	00:00:54
max	tm	157 084		6525		6016		3744		0.073		−0.9		00:00:39	00:00:47
	atso	154 169	−1.86 %	6775	3.83 %	6027	0.18 %	3561	−4.89 %	0.072	−1.37 %	−1.1	−0.2	00:00:39	00:00:46
memctrl	tm	1 180 078		50 560		42 921		30 301		1.508		−1.1		00:01:34	00:02:06
	atso	1 175 283	−0.41 %	50 465	−0.19 %	42 865	−0.13 %	30 157	−0.48 %	1.512	0.27 %	−1.2	0.0	00:01:34	00:02:06
multiplier	tm	1 103 914		43 716		41 015		27 435		0.333		−2.4		00:00:06	00:01:12
	atso	1 124 497	1.86 %	44 946	2.81 %	42 430	3.45 %	26 903	−1.94 %	0.327	−1.80 %	−3.3	−0.9	00:00:07	00:01:07
priority	tm	20 821		922		844		467		0.005		−0.3		00:00:00	00:00:00
	atso	20 821	0.00 %	922	0.00 %	844	0.00 %	467	0.00 %	0.005	0.00 %	−0.3	0.0	00:00:00	00:00:00
router	tm	11 370		525		497		224		0.002		−0.3		00:01:40	00:01:40
	atso	8749	−23.05 %	442	−15.81 %	415	−16.50 %	192	−14.29 %	0.002	0.00 %	−0.3	−0.1	00:01:40	00:01:41
sin	tm	255 985		11 321		9918		6161		0.063		−2.5		00:00:01	00:00:07
	atso	254 125	−0.73 %	11 282	−0.34 %	9884	−0.34 %	6090	−1.15 %	0.061	−3.17 %	−2.6	−0.1	00:00:01	00:00:06
sqrt	tm	1 469 345		60 343		54 343		30 489		0.359		−63.5		00:00:07	00:08:50
	atso	1 328 645	−9.58 %	55 037	−8.79 %	51 743	−4.78 %	26 317	−13.68 %	0.336	−6.41 %	−81.4	−17.9	00:00:07	00:09:47
square	tm	600 679		26 504		24 338		16 783		0.185		−1.0		00:00:01	00:00:06
	atso	597 536	−0.52 %	26 419	−0.32 %	24 248	−0.37 %	16 710	−0.43 %	0.183	−1.08 %	−1.9	−0.9	00:00:01	00:00:07
voter	tm	318 522		13 964		13 093		8391		0.08		−1.1		00:00:01	00:00:03
	atso	318 139	−0.12 %	13 953	−0.08 %	13 079	−0.11 %	8383	−0.10 %	0.072	−10.00 %	−1.1	0.0	00:00:01	00:00:03
Summary	tm	17 639 756		709 439		638 003		421 794						00:08:02	00:54:34
	atso	17 365 066		701 454		633 347		413 092						00:08:03	01:20:03

Table 5.3: Comparison of Algorithm 5.5 without (tm) and with ILP restructuring in line 5 (atso) on EPFL benchmarks.

Chip	Run	Total Area		#Gates before RipOut		#Gates after RipOut		#Gates w\o inv		Netlength		Worst Slack		ILP runtime	Runtime
Chip 1	tm	66 275		12 725		6117		3164		0.031		22.7		00:12:02	00:12:13
	atso	65 421	−1.29 %	12 601	−0.97 %	6221	1.70 %	3044	−3.79 %	0.031	0.00 %	22.7	0.0	00:11:54	00:12:04
Chip 2	tm	152 182		30 180		14 846		7126		0.105		−98.2		00:01:50	00:02:07
	atso	152 350	0.11 %	30 311	0.43 %	15 134	1.94 %	6986	−1.96 %	0.107	1.90 %	−81.9	16.4	00:01:48	00:02:04
Chip 3	tm	261 703		40 671		17 667		8542		0.195		−25.8		00:08:49	00:09:14
	atso	262 022	0.12 %	40 773	0.25 %	17 814	0.83 %	8468	−0.87 %	0.194	−0.51 %	−19.2	6.6	00:08:33	00:08:55
Chip 4	tm	727 539		105 530		39 997		20 199		0.676		−33.3		00:10:56	00:12:03
	atso	731 984	0.61 %	106 636	1.05 %	41 362	3.41 %	20 129	−0.35 %	0.647	−4.29 %	−29.2	4.2	00:10:55	00:12:02
Chip 5	tm	460 603		99 492		46 329		36 130		0.349		−312.0		00:09:22	00:11:09
	atso	466 061	1.18 %	100 573	1.09 %	48 378	4.42 %	35 814	−0.87 %	0.348	−0.29 %	−288.9	23.1	00:09:25	00:11:19
Chip 6	tm	719 784		165 634		60 389		38 951		0.413		−132.6		00:14:41	00:17:33
	atso	738 788	2.64 %	168 786	1.90 %	64 904	7.48 %	38 075	−2.25 %	0.418	1.21 %	−148.5	−15.9	00:14:20	00:16:47
Chip 7	tm	745 571		169 948		67 877		39 760		0.962		−46.4		00:00:04	00:00:57
	atso	745 499	−0.01 %	169 720	−0.13 %	68 138	0.38 %	39 191	−1.43 %	0.966	0.42 %	−45.1	1.4	00:00:04	00:00:53
Chip 8	tm	4 234 678		168 946		77 782		40 619		0.95		−59.2		00:39:56	00:42:15
	atso	4 237 030	0.06 %	168 406	−0.32 %	80 086	2.96 %	40 091	−1.30 %	0.957	0.74 %	−67.9	−8.7	00:40:13	00:42:01
Chip 9	tm	618 006		224 151		78 127		52 327		0.635		−77.0		00:08:12	00:09:32
	atso	617 001	−0.16 %	224 788	0.28 %	79 523	1.79 %	51 540	−1.50 %	0.653	2.83 %	−62.7	14.3	00:08:16	00:09:30
Chip 10	tm	1 401 074		443 664		158 370		104 372		1.547		−82.7		00:09:12	00:13:47
	atso	1 411 213	0.72 %	442 901	−0.17 %	162 572	2.65 %	103 325	−1.00 %	1.58	2.13 %	−89.8	−7.1	00:09:04	00:13:09
Chip 11	tm	10 209 427		161 574		138 988		82 709		4.221		−1491.1		00:29:07	00:33:23
	atso	10 212 296	0.03 %	162 859	0.80 %	140 157	0.84 %	82 079	−0.76 %	4.172	−1.16 %	−1491.1	0.0	00:28:58	00:32:58
Chip 12	tm	11 430 522		694 391		281 496		155 578		5.022		−419.6		00:23:39	00:33:03
	atso	11 444 867	0.13 %	696 233	0.27 %	282 498	0.36 %	153 433	−1.38 %	5.13	2.15 %	−407.5	12.1	00:23:36	00:35:41
Chip 13	tm	14 796 014		1 116 825		421 552		273 534		4.769		−257.1		01:23:57	01:34:18
	atso	14 797 953	0.01 %	1 118 755	0.17 %	427 449	1.40 %	271 109	−0.89 %	4.808	0.82 %	−252.3	4.8	01:24:31	01:34:30
Chip 14	tm	8 811 290		2 914 798		993 736		694 526		20.78		−194.6		02:03:57	05:52:09
	atso	8 898 700	0.99 %	2 929 551	0.51 %	1 019 499	2.59 %	689 638	−0.70 %	20.75	−0.15 %	−197.6	−3.1	02:05:49	05:01:35
Summary	tm	54 634 668		6 348 529		2 403 273		1 557 537						06:15:50	10:43:48
	atso	54 781 185		6 372 893		2 453 735		1 542 922						06:17:34	09:53:33

Table 5.4: Comparison of Algorithm 5.5 without (tm) and with ILP restructuring in line 5 (atso) on chips.

gates but also allows the technology mapping to use less inverters. This is a very natural development. Fewer gates imply a smaller number of nets and therefore fewer inverters are needed. Moreover, fewer nets also usually imply that there is less netlength which one can also see in Table 5.3. Looking at the number of inverters after RipOut, we can see that this number decreases slightly less than the number before RipOut. Apparently, the inverters inserted during technology mapping are distributed slightly more over the different nets than without ILP restructuring.

The results on the chips in Table 5.4 are a bit different. We still see the desired decrease in the number of non-inverter gates. On some chips however, we see an increase in the number of gates before RipOut even though the total number of non-inverter gates decreases on every chip. This increase is thus an increase in the number of inverters. After RipOut, there is a still larger increase that is consistent over the different chips. On most chips, this implies that also the total area increases, which is also measured after RipOut. Apparently, our routine distributes the inverters worse over the instances such that they cannot be ripped out that easily.

In general, technology mapping only inserts one inverter per net in internal nets. It can however insert an inverter behind an output whose net already contains an inverter. Similarly, it can use an input inverted that is an inverter itself. Such nets can thus contain more than one inverter afterward. On the chips, this implies that the solution computed by the ILP requires more internal inverters than when not applying the ILP. Else, we would be able to rip out a similar number of inverters. This might be due to the fact that on chips, we are working on logic that is already very well optimized by lots of logic designers. Thus, it might already be tuned to allow for a technology mapping in which sink parities are usually even. The EPFL instances are optimized less in advance, and thus, not every instance allows for such a technology mapping. This might be the reason why the effect can be seen there only on few chips.

Our flow does not take the position of side-outputs of any gates in the instance into consideration and in general, the netlength can increase for that reason. Similar effects have been experienced for restructuring AND and XOR trees only, too. However, we run the AND or XOR tree optimization after the main part on every instance. Since this contains netlength as an objective, it should almost make up for the increase while restructuring the components. Apart from that, the netlength generally decreases for fewer gates. Thus, we would expect a slight netlength decrease when the number of gates decreases and rather a slight increase or similar values else. This is also what we can see on the EPFL instances on which the number of gates decreases. On the chips, on which the number of gates including inverters is about similar or slightly increasing, we see on average no differences. Thus, the results regarding netlength are as expected.

Regarding the worst slack we do not expect to see good improvements. Our implementation does not yet consider arrival times and hence, we cannot hope to improve with respect to timing consistently. Our ILP solution can sometimes be better and sometimes be worse with respect to timing. This is also the result that we find on the chips: on some chips the timing improves, and on some chips the timing worsens. On the benchmarks, the timing does not change significantly. This is also as expected.

Note that the running time of our base run also includes solving the ILP to decide whether the current instance should be optimized by technology mapping. This is the reason why the running times are about equal for the two runs.

The running times of the ILP solver are very different on the different chips and benchmarks. We have set the running time limit per instance to 100 seconds. There are some larger chips like chip 13 and chip 14 on which solving all the instances takes 1.5 and 2 hours, respectively. This is due to the thousands of instances some of which actually use their time. On chip 13, this is mainly due to about 32 instances that take all of their maximum 100 seconds to be solved. The instances are basically identical and have a maximum tree size of 7. Even though the ILP contains only 11320 variables and 21506 constraints, these seem to be a comparably hard instances. Since they are independent of each other, this chip could benefit a lot from parallelization. Moreover, it might also be beneficial to create a mechanism to detect instances that are equivalent to previous instances and to save their ILP solution and possibly even the result after technology mapping or tree restructuring. Since we wanted to explore how much improvement one can gain with multi-output And functions restructuring, we chose a rather high runtime limit. Nevertheless, there are still some instances that can be improved by choosing an even higher runtime limit. In applications however, one would probably choose a smaller limit though.

Most of the other chips are very fast though. In general, only few instances need more than few seconds. Thus, the solution quality would probably not drop very much when decreasing ILP runtime limit.

The total running time is also quite high for the larger chips and especially for chip 14. This is mainly due to instances with a maximum tree size close to 15. Given the LP solution with millions constraints and variables, it takes some time to determine how exactly the circuit behind the solution looks like. In this case, these are 106 minutes. This running time, just as the number of constraints, is exponential in the maximum size of a trees. Moreover, both the instance collection and the tree restructuring following the technology mapping also take 81 and 31 minutes, respectively.

Relevance of the number of inverters

In general, having many inverters is not desirable because they consume space, power and decrease routability. However, we see an increase in the number of inverters early in the flow of designing a chip. In our algorithm, we do a buffering to see more realistic timing values but our timing model does not take wire delay into account. This implies that repeaters are only placed in nets with high fanout and not in order to speed up signals traveling long distances. Once the chip is in a more optimized state, another buffering is needed that will insert further repeaters. If our inverters are inside nets that are buffered later anyway, these inverters are not harmful to insert now. Thus, it is not clear whether the increase in the number of inverters is indeed something that needs to be fixed or to which extent this needs to be fixed.

Comparing to a solution without technology mapping

For completeness we also want to compare our runs to a version without technology mapping. Thus, our base run (*initial* will in the following not even apply technology mapping and therefore no tree optimization either. It skips lines 3 to 8 of Algorithm 5.5. We compare this run with our flow (*atso*) with respect to the same metrics as before. The results are shown in Table 5.5 and Table 5.6.

The results of these tests look much worse than the previous results which mainly means that the technology mapping does not work as desired. The area increases both on the chips and on the benchmarks. This goes along with a large increase in the total number of gates on the chip after RipOut. For the chips we can see that

Chip	Run	Total Area		#Gates before RipOut		#Gates after RipOut		#Gates w\o inv		Netlength		Worst Slack		ILP runtime	Runtime
adder	initial	48 283		2248		2034		1297		0.028		−0.7		00:00:00	00:00:00
	atso	51 979	7.65 %	2354	4.72 %	2187	7.52 %	1243	−4.16 %	0.028	0.00 %	−1.5	−0.9	00:00:00	00:00:01
arbiter	initial	229 455		9811		8846		5623		0.158		−0.3		00:00:00	00:00:02
	atso	229 455	0.00 %	9811	0.00 %	8846	0.00 %	5623	0.00 %	0.157	−0.63 %	−0.3	0.0	00:00:01	00:00:05
bar	initial	133 192		5315		4903		3600		0.054		−0.3		00:00:00	00:00:00
	atso	133 192	0.00 %	5315	0.00 %	4903	0.00 %	3600	0.00 %	0.054	0.00 %	−0.3	0.0	00:00:00	00:00:01
cavlc	initial	25 155		1103		932		647		0.008		−0.2		00:00:00	00:00:00
	atso	28 952	15.09 %	1221	10.70 %	1093	17.27 %	611	−5.56 %	0.007	−12.50 %	−0.3	0.0	00:00:00	00:00:01
ctrl	initial	3492		157		140		83		0.001		−0.1		00:00:00	00:00:00
	atso	3492	0.00 %	157	0.00 %	140	0.00 %	83	0.00 %	0.001	0.00 %	−0.1	0.0	00:00:00	00:00:00
dec	initial	15 803		655		616		304		0.004		−0.1		00:00:00	00:00:00
	atso	15 803	0.00 %	655	0.00 %	616	0.00 %	304	0.00 %	0.004	0.00 %	−0.1	0.0	00:00:01	00:00:01
hyp	initial	10 444 158		407 030		365 225		251 985		3.163		−215.9		00:00:00	00:06:25
	atso	10 506 654	0.60 %	411 837	1.18 %	370 485	1.44 %	249 166	−1.12 %	3.173	0.32 %	−239.9	−24.0	00:03:34	01:03:07
i2c	initial	47 741		2055		1797		1114		0.027		−0.2		00:00:00	00:00:00
	atso	47 955	0.45 %	2066	0.54 %	1809	0.67 %	1112	−0.18 %	0.026	−3.70 %	−0.2	0.0	00:00:00	00:00:00
int2float	initial	8082		363		322		218		0.002		−0.2		00:00:00	00:00:00
	atso	9546	18.11 %	407	12.12 %	375	16.46 %	200	−8.26 %	0.002	0.00 %	−0.2	0.0	00:00:00	00:00:00
log2	initial	1 327 834		56 377		50 211		32 936		0.561		−4.7		00:00:00	00:00:19
	atso	1 356 072	2.13 %	57 390	1.80 %	51 358	2.28 %	32 370	−1.72 %	0.547	−2.50 %	−5.4	−0.7	00:00:10	00:00:54
max	initial	146 184		6208		5567		3744		0.071		−0.8		00:00:00	00:00:00
	atso	154 169	5.46 %	6775	9.13 %	6027	8.26 %	3561	−4.89 %	0.072	1.41 %	−1.1	−0.2	00:00:39	00:00:46
memctrl	initial	1 171 731		50 143		42 555		30 301		1.504		−1.1		00:00:00	00:00:24
	atso	1 175 283	0.30 %	50 465	0.64 %	42 865	0.73 %	30 157	−0.48 %	1.512	0.53 %	−1.2	0.0	00:01:34	00:02:06
multiplier	initial	1 035 632		41 773		38 680		27 444		0.315		−2.4		00:00:00	00:00:09
	atso	1 124 497	8.58 %	44 946	7.60 %	42 430	9.69 %	26 903	−1.97 %	0.327	3.81 %	−3.3	−0.9	00:00:07	00:01:07
priority	initial	20 821		922		844		467		0.005		−0.3		00:00:00	00:00:00
	atso	20 821	0.00 %	922	0.00 %	844	0.00 %	467	0.00 %	0.005	0.00 %	−0.3	0.0	00:00:00	00:00:00
router	initial	9328		446		424		225		0.002		−0.2		00:00:00	00:00:00
	atso	8749	−6.20 %	442	−0.90 %	415	−2.12 %	192	−14.67 %	0.002	0.00 %	−0.3	−0.1	00:01:40	00:01:41
sin	initial	247 935		11 112		9699		6161		0.062		−2.5		00:00:00	00:00:02
	atso	254 125	2.50 %	11 282	1.53 %	9884	1.91 %	6090	−1.15 %	0.061	−1.61 %	−2.6	−0.1	00:00:01	00:00:06
sqrt	initial	1 265 876		50 620		44 496		30 609		0.315		−61.8		00:00:00	00:00:13
	atso	1 328 645	4.96 %	55 037	8.73 %	51 743	16.29 %	26 317	−14.02 %	0.336	6.67 %	−81.4	−19.6	00:00:07	00:09:47
square	initial	594 106		26 284		24 097		16 783		0.185		−1.0		00:00:00	00:00:03
	atso	597 536	0.58 %	26 419	0.51 %	24 248	0.63 %	16 710	−0.43 %	0.183	−1.08 %	−1.9	−0.9	00:00:01	00:00:07
voter	initial	317 875		13 940		13 068		8391		0.082		−1.1		00:00:00	00:00:01
	atso	318 139	0.08 %	13 953	0.09 %	13 079	0.08 %	8383	−0.10 %	0.072	−12.20 %	−1.1	0.0	00:00:01	00:00:03
Summary	initial	17 092 683		686 562		614 456		421 932						00:00:00	00:07:43
	atso	17 365 066		701 454		633 347		413 092						00:08:03	01:20:03

Table 5.5: Comparison between our routine and initial values on EPFL benchmarks.

Chip	Run	Total Area		#Gates before RipOut		#Gates after RipOut		#Gates w\o inv		Netlength		Worst Slack		ILP runtime	Runtime
Chip 1	initial	63 658		12 355		5552		3115		0.03		22.7		00:00:00	00:00:01
	atso	65 421	2.77 %	12 601	1.99 %	6221	12.05 %	3044	−2.28 %	0.031	3.33 %	22.7	0.0	00:11:54	00:12:04
Chip 2	initial	143 708		29 475		12 626		7029		0.101		−85.0		00:00:00	00:00:02
	atso	152 350	6.01 %	30 311	2.84 %	15 134	19.86 %	6986	−0.61 %	0.107	5.94 %	−81.9	3.1	00:01:48	00:02:04
Chip 3	initial	259 237		40 424		17 222		8531		0.193		−26.6		00:00:00	00:00:03
	atso	262 022	1.07 %	40 773	0.86 %	17 814	3.44 %	8468	−0.74 %	0.194	0.52 %	−19.2	7.4	00:08:33	00:08:55
Chip 4	initial	718 156		104 466		38 317		20 128		0.66		−25.6		00:00:00	00:00:09
	atso	731 984	1.93 %	106 636	2.08 %	41 362	7.95 %	20 129	0.00 %	0.647	−1.97 %	−29.2	−3.5	00:10:55	00:12:02
Chip 5	initial	448 242		97 867		43 269		36 031		0.334		−327.5		00:00:00	00:00:12
	atso	466 061	3.98 %	100 573	2.76 %	48 378	11.81 %	35 814	−0.60 %	0.348	4.19 %	−288.9	38.6	00:09:25	00:11:19
Chip 6	initial	638 696		163 868		45 739		38 245		0.376		−164.6		00:00:00	00:00:19
	atso	738 788	15.67 %	168 786	3.00 %	64 904	41.90 %	38 075	−0.44 %	0.418	11.17 %	−148.5	16.1	00:14:20	00:16:47
Chip 7	initial	719 863		167 669		63 398		39 259		0.955		−46.7		00:00:00	00:00:19
	atso	745 499	3.56 %	169 720	1.22 %	68 138	7.48 %	39 191	−0.17 %	0.966	1.15 %	−45.1	1.6	00:00:04	00:00:53
Chip 8	initial	4 201 950		164 488		71 045		39 846		0.935		−60.3		00:00:00	00:00:18
	atso	4 237 030	0.83 %	168 406	2.38 %	80 086	12.73 %	40 091	0.61 %	0.957	2.35 %	−67.9	−7.6	00:40:13	00:42:01
Chip 9	initial	562 842		220 965		68 284		51 512		0.609		−63.8		00:00:00	00:00:23
	atso	617 001	9.62 %	224 788	1.73 %	79 523	16.46 %	51 540	0.05 %	0.653	7.22 %	−62.7	1.1	00:08:16	00:09:30
Chip 10	initial	1 268 735		429 331		135 124		100 823		1.491		−72.7		00:00:00	00:01:11
	atso	1 411 213	11.23 %	442 901	3.16 %	162 572	20.31 %	103 325	2.48 %	1.58	5.97 %	−89.8	−17.1	00:09:04	00:13:09
Chip 11	initial	10 216 160		160 566		138 155		82 929		4.162		−1491.1		00:00:00	00:00:19
	atso	10 212 296	−0.04 %	162 859	1.43 %	140 157	1.45 %	82 079	−1.02 %	4.172	0.24 %	−1491.1	0.0	00:28:58	00:32:58
Chip 12	initial	11 247 754		685 018		244 886		153 033		4.928		−330.7		00:00:00	00:05:13
	atso	11 444 867	1.75 %	696 233	1.64 %	282 498	15.36 %	153 433	0.26 %	5.13	4.10 %	−407.5	−76.8	00:23:36	00:35:41
Chip 13	initial	14 695 185		1 108 233		401 498		271 717		4.727		−254.6		00:00:00	00:05:11
	atso	14 797 953	0.70 %	1 118 755	0.95 %	427 449	6.46 %	271 109	−0.22 %	4.808	1.71 %	−252.3	2.3	01:24:31	01:34:30
Chip 14	initial	8 462 186		2 903 386		922 860		693 392		18.98		−187.9		00:00:00	00:30:49
	atso	8 898 700	5.16 %	2 929 551	0.90 %	1 019 499	10.47 %	689 638	−0.54 %	20.75	9.34 %	−197.6	−9.7	02:05:49	05:01:35
Summary	initial	53 646 372		6 288 111		2 207 975		1 545 590						00:00:00	00:44:33
	atso	54 781 185		6 372 893		2 453 735		1 542 922							09:53:33

Table 5.6: Comparison between our routine and initial values on chips.

there is a large difference between the increases in the number of gates before and after ripping out the inverters. On some chips there is a hugh increase compared to the initial run of the number of gates after RipOut. For chip 5 this is even 39.2%. In contrast, the increase in the number of gates is much less before ripping out inverters. On chip 5 this is only by 2.83%. Since we did not see that much increase when comparing to a base run with technology mapping, this implies that most of the increase is due to the technology mapping itself. Not only our routine has a bad impact on the number of repeaters that can be ripped out but already performing technology mapping at all has a bad impact.

Since we have not worked on the technology mapping in the context of this thesis, we do not want to elaborate this further here.

5.7 Conclusion and future work

The decreases in the number of operations that we have seen in Table 5.2 looks very promising. Moreover, we have also seen in example instances that the algorithm indeed works as expected. When incorporating this into a flow into the chip this yields very good results on the benchmarks and mixed results on the chips. Apparently, one can indeed save some (non-inverter) gates and area even though the logic on our chips was preoptimized by many different industrial algorithms. Already reducing the number of gates a little usually helps a lot in later stages since this leads to better netlength and congestion values which in turn allow for further improvements. We have seen that the number of non-repeater gates can indeed be reduced by considering several AND or XOR functions at once. This justifies that looking at this problem is indeed an approach that can be useful in the modern chip design. Thus, it is worth to put further effort into this approach and see how one can use this restructuring. The increasing number of inverters after ripping out all but one inverter is something one should keep in mind. However, further testing is required to see how this restructuring behaves after a buffering that does not only buffer nets with high fanout but all nets in a timing model with wire delays. Such tests should not only add a buffering in more realistic timing model but needs to do a lot of other timing optimization first to see the actual impact. If there are repeaters inserted into most nets anyway, it might not be too important to avoid single inverters in an early stage of the optimization.

Of course there are different future improvements remaining. This was merely a first overall test to see what can be gained by restructuring multi-output AND functions on real world instances.

Improvements in technology mapping

Probably the most important step is to improve the technology mapping to use less inverters. We have already previously encountered inverter problems in our technology mapping. When restricting to one candidate at vertices with at least two successors, we do not yet know for which predecessors we build one inverter. We do not make a common choice for candidates succeeding a single vertex whether to use an inverter or not. To allow for such a common choice is however very difficult because every candidate has multiple implications on predecessors that would all need to be taken into account. Looking at the results of Armbruster [Arm21], who has tested this routine on different instances, also shows fewer increases in the number of gates. Thus, the increase could also be related to technology mapping running in virtual timing and us measuring in a timing model without wire delay. This technology mapping always tries to fulfill the required arrival times, and this might lead to a

larger increase in the number of inverters than in virtual timing. Hence, it would make sense to have a suitable internal timing model in technology mapping or to be more flexible with respect to required arrival times.

Furthermore, if one considers it important that there are only few inverters after ripping out all but one inverter per net, one should investigate why there are many more inverters distributed over the different nets after the ILP application. Possibly, the cost used during technology mapping can be adapted to avoid some internal inverters.

Moreover, there is another feature that might be helpful in technology mapping. Consider an instance in which two Nand3 gates share two inputs. One could first combine these two inputs and then use two Nand2 gates to compute the needed results. If a Nand2 gate is around the same size as a Nand3 though, this might not always make sense. Thus, technology mapping should be able to consider merging the first And2 into both its successors. Even though this is an improvement in the number of 2-ary gates, technology mapping should be able to realize that it is worth merging the two-output gates into both their successors.

Improving the ILP

Apart from technology mapping, it could also make sense to improve on the ILP itself. So far, the only objective is to minimize the number of 2-ary gates. To use it on timing-critical instances, too, one would need to incorporate a delay or depth model into the ILP.

We show a formulation which includes the depth of the graph as well (and this can analogously be generalized to arrival times). To do so, we can modify our variables and add multiple versions for every set S that we might want to build. We want $x_S^d = 1$ if and only if set S is built with depth at most d. We use the notation x_S^∞ for the variable x_S^d with maximum d. Then we require that every set that is built with depth at most d can be partitioned into two subsets that are build with depth at most $d-1$. This results in the following integer linear program.

$$
\begin{aligned}
\min \quad & \sum_{S \in \mathcal{S}_2} x_S^\infty & & & \\
s.t. \quad & \frac{1}{2} \sum_{\emptyset \subsetneq S' \subsetneq S} \min\{x_{S'}^{d-1}, x_{S \setminus S'}^{d-1}\} & \geq & x_S^d & \forall\, S \in \mathcal{S}_2, d \in [1, \ldots, |S|-1] \\
& x_S^d & \geq & x_S^{d-1} & \forall\, S \in \mathcal{S}_2, d \in [1, \ldots, |S|-1] \\
& x_T & = & 1 & \forall\, T \in \mathcal{T} \\
& x_S^d & \in & \{0,1\} & \forall\, S \in \mathcal{S}, d \in [0, \ldots, |S|-1]
\end{aligned}
\tag{5.1}
$$

Apart from arrival times, one could also consider to add a netlength approximation model to the ILP.

This change of objective probably needs to go along with a speedup or a reduction in the number of variables. There are certainly many variables in the ILP that are not needed since they correspond to a decomposition into two sets that both cannot be reused by more than tree. For every tree, it suffices to keep one such decomposition and one could get rid of several decomposition and set variables in that fashion. It is however not directly clear how to determine such a set fast and reliably.

Application to And-inverter graphs

We have tested this optimization routine based on a delay model without wire length because we assume that it would serve best to apply it very early before a reasonable

placement exists. Results have shown that there are some issues with technology mapping and the numbers of inverters inserted thereby. However, we have also seen that one can gain certainly in the number of 2-ary gates using this routine. Thus, it would be extremely interesting to see how this approach performs when being applied during logic synthesis. In this initial step, Boolean function optimization is applied independently of the technology. Often, this synthesis step partially works on AND-inverter graphs in which all logic is represented by AND and inverter gates. On such graphs, the described problem of internal inverters within our components does not apply and therefore, such tests sound very promising.

Summary

In this thesis, we have studied different approaches for restructuring different Boolean functions. We started by considering the addition of two integers as one of the most classical tasks in this area. This task reduces to computing the And-Or paths that correspond to the carry bits. To achieve fast adders, it is necessary to compute them with a small depth. And-Or paths have been subject to substantial prior research, for example by Grinchuk [Gri09] and Brenner and Silvanus [BS24]. By investigating their algorithm along with their analysis in joint work with Ulrich Brenner, we improved the upper bound on the optimum depth from $\log_2 m + \log_2 \log_2 m + 0.65$ to $\log_2 m + \log_2 \log_2 m$, which is tight for small And-Or paths. Moreover, we were able to improve this bound asymptotically to $\log_2 m + \log_2 \log_2 m - 1$ and have shown a bound on the rate of convergence, too. The lower bounds by Commentz-Walter [Com79] and Dietz [Die24] imply that our result cannot be improved further by more than an additive constant of -2. Our result was achieved by improving the bound on how many alternating inputs can be part of a circuit of a certain depth with a certain number of symmetric inputs. Apart from being able to increase the scaling factor by refining the analysis, we discovered that a better bound can be achieved when increasing the scaling factor depending on the depth.

We showed how to linearize our circuits by using a framework for multi-output And functions that was adapted from Brenner and Silvanus [BS24]. A first analysis showed that this leads to a bound of $4.08m$. We were able to improve this size bound by checking actual sizes for small circuits of depth up to 16. This allowed us to improve the bound on our size to $2.63m$ instead.

Using these linear size And-Or paths allowed us to achieve the fastest known linear size adders. We followed the recursion strategy of Brenner and Silvanus [BS24]. To achieve a smaller depth bound, we needed to replace known adders for small input numbers by adders implied our new And-Or paths though. In that fashion, we were able to compute adder circuits with depth at most $\log_2 n + \log_2 \log_2 n + \log_2 \log_2 \log_2 n + 5.6$ and size at most $19n$.

Since restructuring multi-output And functions was a key ingredient for linear size And-Or paths and thus also linear size adders, we have considered this problem in a more general form in Chapter 4. We proved that this is already APX-hard when restricting to functions with at most three input variables by reducing the problem to vertex cover. To approximate this problem, we introduced a linear program to model it. Using a randomized rounding approach, we were able to give a 1.212- and a 1.731-approximation for the cases with three and four variables per function, respectively.

We have seen that these algorithms can be derandomized and work better than combinatorial algorithms we showed. Moreover, we gave a general combinatorial algorithm that works for arbitrary numbers of inputs and gives a $\frac{2}{3}k$-approximation where k is the maximum number of inputs of an AND function. Our model allowed to transfer these results to OR and XOR functions, too.

Last, we considered the problem of multi-output AND or XOR function optimization on practical instances. We decomposed the whole chip into gates with two inputs and applied De Morgan rules to make it consist of ANDs, XORs and inverters only. We then collected single-output AND and single-output XOR functions implemented on the chip and combined them to a larger instance whenever they shared at least two inputs. Afterward, we introduced an integer linear program similar to the rounded linear program introduced in our theoretical work and solved it mostly exactly using an ILP solver to decrease the number of gates in the instances. This algorithm was then followed by a technology mapping and some other restructuring routines. The tests we ran showed a significant improvement in the number of 2-ary gates in our optimization due to our algorithm. Apart from an increase in the number of inverters on the chips, we have seen that on both the benchmarks and the chips, the numbers of logic gates after technology mapping decreased. This shows that our approach is reasonable and can be used to improve the chip in early stages of optimization.

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 pp. 9, 122).

[Arm21] Susanne Armbruster. "Directed Acyclic Graph Partitioning and Applications in Technology Mapping, Masterarbeit". MA thesis. University of Bonn, 2021 (cit. on pp. 119, 136).

[Arm24] Susanne Armbruster. *Size Minimization For Multi-Output AND-Functions*. 2024 (cit. on pp. 77, 82).

[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. 111).

[BH19] Ulrich Brenner and Anna Hermann. "Faster Carry Bit Computation for Adder Circuits with Prescribed Arrival Times". In: *ACM Trans. Algorithms* 15.4 (2019), Art. No. 45 (cit. on p. 26).

[BK10] Nikhil Bansal and Subhash Khot. "Inapproximability of Hypergraph Vertex Cover and Applications to Scheduling Problems". In: *Automata, Languages and Programming*. 2010, pp. 250–261 (cit. on p. 95).

[BK82] R. P. Brent and H. T. Kung. "A regular layout for parallel adders". In: *Transactions on Computers* C-31.3 (1982), pp. 260–264 (cit. on p. 21).

[BM10] Robert Brayton and Alan Mishchenko. "ABC: An Academic Industrial-Strength Verification Tool". In: *Computer Aided Verification*. 2010, pp. 24–40 (cit. on p. 78).

[BS23] Ulrich Brenner and Anna Silvanus. "BonnLogic: Delay optimization by And-Or Path restructuring". In: *Integration* 89 (2023), pp. 123–133 (cit. on pp. 20, 28, 110).

[BS24] Ulrich Brenner and Anna Silvanus. *Faster Linear-Size And-Or Path and Adder Circuits*. 2024 (cit. on pp. 8, 22, 24, 26, 32, 33, 49, 51–53, 57, 62, 66, 67, 70, 71, 139).

[BSS22] Ulrich Brenner, Anna Silvanus, and Jannik Silvanus. "Constructing depth-optimum circuits for adders and And-Or paths". In: *Discrete Applied Mathematics* 310 (2022), pp. 10–31 (cit. on p. 32).

[BU11] David Buchfuhrer and Christopher Umans. "The complexity of boolean formula minimization". In: *Journal of Computer and System Sciences* 77.1 (2011), pp. 142–153 (cit. on pp. 8, 79).

[CC06] Miroslav Chlebík and Janka Chlebíková. "Complexity of approximating bounded variants of optimization problems". In: *Theoretical Computer Science* 354.3 (2006), pp. 320–338 (cit. on p. 84).

[CH11] Yves Crama and Peter L. Hammer. *Boolean Functions: Theory, Algorithms, and Applications*. 2011 (cit. on p. 11).

[CKS17] Chang Chual, R.B.N. Kumarl, and B. Sireesah. "Design and analysis of low-power and area efficient N-bit parallel binary comparator". In: *Analog Integrated Circuits and Signal Processing* 92 (2017), pp. 225–231 (cit. on p. 19).

[Com79] Beate Commentz-Walter. "Size-depth tradeoff in monotone Boolean formulae". In: *Acta Informatica* 12 (1979), pp. 227–243 (cit. on pp. 22, 139).

[Dar+81] John A. Darringer, William H. Joyner, C. Leonard Berman, and Louise Trevillyan. "Logic Synthesis Through Local Transformations". In: *IBM Journal of Research and Development* 25.4 (1981), pp. 272–280 (cit. on p. 78).

[Die24] F. Dietz. "Untere Schranken für Tiefe von verallgemeinerten AND-OR-Pfaden". German. BA thesis. University of Bonn, 2024 (cit. on pp. 22, 23, 139).

[Din+05] Irit Dinur, Venkatesan Guruswami, Subhash Khot, and Oded Regev. "A New Multilayered PCP and the Hardness of Hypergraph Vertex Cover". In: *SIAM Journal on Computing* 34.5 (2005), pp. 1129–1146 (cit. on p. 95).

[DS16] Evrim Dalkiran and Hanif D Sherali. "RLT-POS: Reformulation-Linearization Technique-based optimization software for solving polynomial programming problems". In: *Mathematical Programming Computation* 8.3 (2016), pp. 337–375 (cit. on p. 79).

[Elb17] Lucas Elbert. "Approximationsalgorithmen im Technology Mapping". German. BA thesis. University of Bonn, 2017 (cit. on p. 119).

[Elm48] William 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. 109).

[EV23] Sourour Elloumi and Zoé Verchère. "Efficient linear reformulations for binary polynomial optimization problems". In: *Computers & Operations Research* 155 (2023), p. 106240 (cit. on pp. 78, 79).

[For60] Robert Fortet. "Applications de l'algebre de Boole en recherche opérationelle". In: *Trabajos de Estadistica* 11 (2 1960), pp. 111–118 (cit. on p. 79).

[Gei12] Laura Geisen. "Optimierung symmetrischer Funktionen im Chip Design". German. MA thesis. University of Bonn, 2012 (cit. on pp. 79, 110, 111).

[Gök14] Alexander Göke. "Approximationsalgorithmen für symmetrische Funktionen". German. MA thesis. University of Bonn, 2014 (cit. on p. 110).

[Gol76] Martin C. Golumbic. "Combinatorial merging". In: *Transactions on Computers* C-25.11 (1976), pp. 1164–1167 (cit. on pp. 79, 111).

[Gri09] Mikhail 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. 22, 26, 28, 32, 139).

[Hal02] Eran Halperin. "Improved Approximation Algorithms for the Vertex Cover Problem in Graphs and Hypergraphs". In: *SIAM Journal on Computing* 31.5 (2002), pp. 1608–1623 (cit. on p. 95).

[Hel+11] Stephan Held, Bernhard Korte, Dieter Rautenbach, and Jens Vygen. "Combinatorial optimization in VLSI design." In: *Combinatorial Optimization* 31 (2011), pp. 33–96 (cit. on p. 107).

[Hel63] Leo Hellerman. "A Catalog of Three-Variable Or-Invert and And-Invert Logical Circuits". In: *Transactions on Electronic Computers* EC-12.3 (1963), pp. 198–223 (cit. on p. 78).

[Her20] Anna Hermann. "Faster Circuits for And-Or Paths and Binary Addition". PhD thesis. University of Bonn, 2020 (cit. on pp. 11, 15, 22, 28, 31, 32, 66–68, 70, 71, 119).

[Hit18] Jan Maik Hitzschke. "Untere Schranken für Tiefe und Delay von AND-OR-Pfaden". German. BA thesis. University of Bonn, 2018 (cit. on p. 22).

[HK17] Stephan Held and Nicolas Kämmerling. "Two-level rectilinear Steiner trees". In: *Computational Geometry* 61 (2017), pp. 48–59 (cit. on p. 79).

[HR13] Stephan Held and Daniel Rotter. "Shallow-Light Steiner Arborescences with Vertex Delays". In: *Integer Programming and Combinatorial Optimization*. 2013, pp. 229–241 (cit. on p. 111).

[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), Art. No. 4 (cit. on p. 22).

[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 p. 21).

[ILO20] Rahul Ilango, Bruno Loff, and Igor C. Oliveira. "NP-Hardness of Circuit Minimization for Multi-Output Functions". In: *35th Computational Complexity Conference (CCC 2020)*. Vol. 169. 2020, 22:1–22:36 (cit. on pp. 16, 78, 79).

[Käm12] Nicolas Kämmerling. "Physikalisches Layout symmetrischer Funktionen". German. MA thesis. University of Bonn, 2012 (cit. on p. 110).

[Kar72] Richard M. Karp. "Reducibility among Combinatorial Problems". In: *Complexity of Computer Computations*. 1972, pp. 85–103 (cit. on p. 82).

[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. 22, 25, 28).

[KRV07] Bernhard Korte, Dieter Rautenbach, and Jens Vygen. "BonnTools: Mathematical innovation for layout and timing closure of systems on a chip". In: 95.3 (2007), pp. 555–572 (cit. on p. 107).

[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* C-22.8 (1973), pp. 786–793 (cit. on pp. 21, 24).

[Lev73] Leonid A Levin. "Universal sequential search problems". In: *Problems of information transmission* 9.3 (1973), pp. 265–266 (cit. on p. 79).

[Ley22] C. Ley. "Vereinfachte Korrektur von Grinchuks Beweis zur Tiefe von And-Or-Pfaden". German. BA thesis. University of Bonn, 2022 (cit. on p. 22).

[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. 21, 69, 70, 74).

[Li+10] Zhuo Li, David A. Papa, Charles J. Alpert, Shiyan Hu, Weiping Shi, Cliff Sze, and Ying Zhou. "Ultra-fast interconnect driven cell cloning for minimizing critical path delay". In: *Proceedings of the 19th International Symposium on Physical Design*. 2010, pp. 75–82 (cit. on p. 122).

[Lu+24] Zhenjian Lu, Noam Mazor, Igor C. Oliveira, and Rafael Pass. *Lower Bounds on the Overhead of Indistinguishability Obfuscation*. Cryptology ePrint Archive, Paper 2024/1524. 2024 (cit. on p. 79).

[Nöb21] Christian Nöbel. "Extending BonnLogic to Multiple Outputs". MA thesis. University of Bonn, 2021 (cit. on pp. 79, 110).

[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 p. 21).

[Par79] Stott Parker. "Combinatorial Merging and Huffman's Algorithm". In: *Transactions on Computers* C-28.5 (1979), pp. 365–367 (cit. on p. 79).

[PR90] L.T. Pillage and R.A. Rohrer. "Asymptotic waveform evaluation for timing analysis". In: *Transactions on Computer-Aided Design of Integrated Circuits and Systems* 9.4 (1990), pp. 352–366 (cit. on p. 109).

[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 p. 109).

[RPH83] J. Rubinstein, P. Penfield, and M.A. Horowitz. "Signal Delay in RC Tree Networks". In: *Transactions on Computer-Aided Design of Integrated Circuits and Systems* 2.3 (1983), pp. 202–211 (cit. on p. 109).

[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 p. 21).

[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. 21).

[Sav98] John E. Savage. *Models of Computation*. Vol. 136. 1998 (cit. on p. 11).

[Skl60] J. Sklansky. "Conditional-sum addition logic". In: *IRE Transactions on Electronic computers* EC-9.2 (1960), pp. 226–231 (cit. on p. 21).

[Sun+22] Sahithi Sunkireddy, Sahithi Thallapally, Ganesh Avula, and B. Shivani Gayatri Devi. "Design of High Speed and Area Efficient N-Bit Digital Comparator". In: *International Journal for Research in Applied Science & Engineering Technology* 10 (11 2022), pp. 1522–1527 (cit. on p. 19).

[TP20] Piyush Tyagi and Rishikesh Pandey. "High-speed and area-efficient scalable N-bit digital comparator". In: *IET Circuits, Devices & Systems* 14.4 (2020), pp. 450–458 (cit. on p. 19).

[Weg87] Ingo Wegener. *The Complexity of Boolean Functions*. 1987 (cit. on p. 21).

[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. 20, 109).

[Xia+10] Hua Xiang, Haoxing Ren, Louise Trevillyan, Lakshmi Reddy, Ruchir Puri, and Minsik Cho. "Logical and Physical Restructuring of Fan-in Trees". In: *Proceedings of the 19th International Symposium on Physical Design*. 2010, pp. 67–74 (cit. on p. 79).

[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. 21).