Positive and Negative Proofs for Circuits and Branching Programs

Olga Dorzweiler, Thomas Flamm, Andreas Krebs, and Michael Ludwig

WSI - University of Tübingen, Germany, Sand 13, 72076 Tübingen, Germany
{dorzweiler,flamm,krebs,ludwigm}@informatik.uni-tuebingen.de

Abstract. We extend the # operator in a natural way and derive a new type of counting complexity. While $\#\mathcal{C}$ classes (where $\mathcal{C}$ is some circuit-based class like $\mathbf{NC^1}$) only count proofs for acceptance of some input in circuits, one can also count proofs for rejection. The here proposed Zap-$\mathcal{C}$ complexity classes implement this idea. We show that Zap-$\mathcal{C}$ lies between $\#\mathcal{C}$ and Gap-$\mathcal{C}$. In particular we consider Zap-$\mathbf{NC^1}$ and polynomial size branching programs of bounded and unbounded width. We find connections to planar branching programs since the duality of positive and negative proofs can be found again in the duality of graphs and their co-graphs. This links to possible applications of our contribution, like closure properties of complexity classes.

1 Introduction

Besides Turing machines, circuits are a well studied model of computation for the study of low level complexity classes. Measures of complexity in circuits include depth and the number of gates which are roughly speaking an analogue to time and space complexity in Turing machines. When regarding parallels between Turing machines and circuits, a natural question is, what the counterpart to non-determinism in circuits is. A non-deterministic Turing machine can have more than one accepting computation on some input. In fact, the counterpart to the presence of multiple accepting computations is the presence of proof trees in circuits. A proof tree is a sub-tree of the tree unfolding of a circuit, which is a witness for acceptance of some input word. When looking at the circuit-based characterization of, say $\mathbf{NP}$, one can observe that the number of accepting computations and the number of proof trees coincide [Ven92].

How can we calculate the number of proof trees in a circuit? It can be verified easily that if we move to an arithmetic interpretation of the circuit, it computes the number of proof trees [VT89]. I.e. we interpret And as $\times$ and Or as $+$. Since we cannot treat negation in this setting directly, we assume w.l.o.g. the circuit to be monotone. If we want to address functions counting proof trees in circuits (or equivalently arithmetic circuits) of some complexity bound, say $\mathbf{NC^1}$, we write $\#\mathbf{NC^1}$.

It is an open question whether $\#\mathcal{C}$ functions are closed under subtraction. Here, the case we are most interested in is $\mathcal{C} = \mathbf{NC^1}$. This motivated another type of counting complexity: Gap. Where $\#\mathcal{C}$ functions range over non-negative

H. Jürgensen et al. (Eds.): DCFS 2014, LNCS 8614, pp. 270–281, 2014.
© Springer International Publishing Switzerland 2014

integers, GAP-$\mathcal{C}$ functions range over integers. GAP-$\mathcal{C}$ functions are realized by arithmetic circuits with gates of types $\{+, -, \times\}$. By [FFK94, All04] we know that GAP-$\mathcal{C}$ = $\#\mathcal{C} - \#\mathcal{C}$, what motivated its naming. That means that each GAP function can be computed by an arithmetic circuit only having a single subtraction gate.

Boolean circuits can also compute arithmetic functions. Such a circuit has as many output gates as necessary to display the result integer in binary representation. Hence one can ask e.g. if $\mathbf{NC^1} = \#\mathbf{NC^1}$ or even $\mathbf{NC^1}$ = GAP-$\mathbf{NC^1}$. By Jung [Jun85] we know, that those classes lie extremely close, but it is still unknown if they coincide.

At this point our contribution comes into play. We propose a new type of counting complexity which fits in between $\#\mathcal{C}$ and GAP-$\mathcal{C}$ very naturally. The starting point for our definition is the observation that we can extend the notion of a proof tree. A (now called positive) proof tree is a witness for a word being accepted. If a word is rejected, there are also witnesses: negative proof trees. To our knowledge, negative proof trees haven't been considered before even though the duality of positive and negative proofs is appealing. We call[1] our new counting complexity classes ZAP-$\mathcal{C}$. ZAP-$\mathcal{C}$ functions are of the form $\Sigma^* \to \mathbb{Z} \setminus \{0\}$. The image is positive iff there are positive proof trees and negative in the case of the existence of negative proof trees.

Providing the base of the ZAP definition we show an arithmetic interpretation of circuits which then calculate the corresponding ZAP-$\mathcal{C}$ function. By the nature of ZAP, we are not restricted to monotone circuits any more in contrast to the # case. The second interesting result is that in the case of $\mathcal{C} \in \{\mathbf{NC^i}, \mathbf{AC^i} | i \geq 0\}$ the ZAP-$\mathcal{C}$ functions can be written as differences of $\#\mathcal{C}$ functions with the restriction that the result must not be 0. This uses the fact that each circuit can be transformed in a way that each input has exactly either one negative or one positive proof tree. Those two results place ZAP-$\mathcal{C}$ right between $\#\mathcal{C}$ and GAP-$\mathcal{C}$. So ZAP might give us new possibilities to examine the differences between $\#\mathcal{C}$ and GAP-$\mathcal{C}$.

ZAP circuits are the one major topic of this work. The other one is (nondeterministic) ZAP branching programs (BP). Starting point is the celebrated work of Barrington which showed that bounded width polynomial size branching programs (BWBP) are equally powerful as $\mathbf{NC^1}$ circuits. In the case of BPs we are also interested in counting. In BPs a witness for acceptance is a path from source to target. By [CMTV98] we have that the task of counting paths can be expressed as matrix multiplication which is possible in $\#\mathbf{NC^1}$. However we do not know if counting proof trees in $\mathbf{NC^1}$ circuits is possible in $\#\mathbf{BWBP}$ hence $\#\mathbf{NC^1} \stackrel{?}{=} \#\mathbf{BWBP}$ is an open question.

We extend the ZAP idea to BPs. To do so, we need an analogon to the negative proof trees known from circuits. We found this in the notion of *cuts*. A cut in our sense is a partition of the BP's nodes in two, so that source and target are separated and no undesired edge goes between the two parts. From this it is clear that given a BP and some input there is a path iff there is no cut. In the

[1] The naming is motivated by the set of integers $\mathbb{Z}$ and GAP.

case of circuits, positive and negative proofs are dual by negating the circuit. This in a way is inherited by BPs. If we have a planar BP, the counter part to negation is moving to the dual graph. The number of paths becomes the number of cuts and vice versa, so we have switched the sign of the function. We show how ZAP BPs are related to ZAP-$\mathbf{NC^1}$. We have a construction to simulate ZAP-$\mathbf{NC^1}$ functions with BPs. The BPs generated that way are planar but not bounded. This raises questions concerning boundedness and planarity in BPs and ZAP-$\mathbf{NC^1}$ whose answers could give insights in the $\#\mathbf{NC^1} \stackrel{?}{=} \#\mathbf{BWBP}$ problem.

The paper is structured as follows: We begin with a *Preliminaries* section providing all definitions necessary concerning computational complexity, circuits, arithmetic circuits and BPs. The following section, *Results*, summarizes and states our results. In the subsequent sections we give proofs for the theorems. In the end we give an overview and outlook. Two proofs can be found in the *Appendix* to comply with the length constraint.

We thank the anonymous referees for their helpful comments.

2 Preliminaries

In this paper words are always built from the alphabet $\Sigma = \{0, 1\}$. By $\mathbb{Z}$ we denote integers and by $\mathbb{N}$ the non-negative integers. We assume the reader to be acquainted with Turing Machines and elementary complexity classes like **L**, **NL**, **P** and **NP**. See e.g. [Pap94] for basics in computational complexity.

A Turing machine M accepts a word w, if there is a computation on the input resulting in an accepting state. For non-deterministic machines, there can be more than one. We denote the number of accepting computations by $\#acc_M(w)$. The set of functions $\#acc_M : \Sigma^* \to \mathbb{N}$ for Turing machines M in the bound of some complexity class, say **NP**, is denoted by #**P**. Similarly we get #**L** from **NL**.

A circuit C is a connected acyclic graph with a designated output node and n input nodes, where n in the in-degree of the circuit. All other nodes are assigned Boolean functions like AND, OR and NEGATION. There are also other gates, e.g. threshold gates or modulo gates, but those are not in the scope of this work.

A word is accepted by a circuit if the computation results to 1. Since a circuit can only treat words of some constant length, we need to speak about circuit families: $(C_n)_{n\in\mathbb{N}}$. If such a family is computable in the limits of some complexity class, we speak of a uniform circuit family, e.g. DLOGTIME-uniformity. If we bound circuit families in the number of gates, depth or fan-in, we get circuit based complexity classes like $\mathbf{AC^0}$, $\mathbf{ACC^0}$, $\mathbf{TC^0}$ and $\mathbf{NC^1}$. The latter one corresponds to circuits of logarithmic depth, polynomial size and a limit of the fan-in of gates of two. Check e.g. [Vol99] for basics in circuit complexity.

Circuits made up of AND and OR gates can be transformed in a *negation normal form*. We duplicate the input nodes, so that each one has a negated twin. By repeated application of DeMorgan's law, we get a monotone circuit computationally equivalent to the original one, i.e. without negation gates (only negated input gates may be present).

If we take a monotone circuit and replace AND by $\times$ and OR by $+$, then we have arithmetized the circuit, i.e. a circuit which operates over the semi-ring $(\mathbb{N}, +, \times, 0, 1)$. It now computes a function $\Sigma^n \to \mathbb{N}$. The set of functions $\Sigma^* \to \mathbb{N}$ generated by the arithmetic interpretation circuit families in some complexity class $\mathcal{C}$ is denoted as $\#\mathcal{C}$. This way we get e.g. $\#\mathbf{AC^0}$ and $\#\mathbf{NC^1}$. Due to [Ven92] we know that this is consistent with counting complexity as it is defined for Turing machines: If we take the monotone circuit characterization of **NP**, then counting accepting computations is the same as arithmetizing the corresponding circuits. It is noteworthy that this is not the only way to arithmetize. In [Bei93], Beigel surveys different possibilities.

Also, we know by [VT89] that arithmetic circuits count proofs trees, which can be seen as levels of acceptance in circuits. Unfortunately this tells us nothing about the efficiency of a circuit since the absence of many proof trees does not imply the absence of redundancy in the circuit structure [Fla12]. Given a circuit C, we can unfold it by iteratively treating gates with fan-out greater than one. If g is such a gate with fan-out $k > 1$, let A be the sub-circuit of C, whose output gate is the output of g. By adding $k-1$ more duplicates of A, we can build an equivalent circuit, where g and its duplicates only have fan-out one. Finally we get an equivalent circuit which is a tree. The number of sub-trees whose root is the circuit's output gate and which result to 1 is the number of proof trees.

In # classes the closure under (modified) subtraction is unknown which motivated another type of arithmetic class. If we consider arithmetic circuits over $\mathbb{Z}$ with gates of types in $\{+, -, \times\}$, we get GAP complexity classes [FFK94]. A function f is in GAP-$\mathcal{C}$ iff there are $g, h \in \#\mathcal{C}$ so that $f = g - h$.

If we want to compare # and GAP classes, we do this on non-negative output values. We can even compare those counting classes with Boolean classes. In this case the Boolean circuit has to have several outputs computing the bits of the binary representation of the resulting integer. For a survey on arithmetic circuit complexity, see the survey of Allender [All04].

Another model of computation are non-deterministic branching programs (BP). Note that counting in deterministic BPs is trivial. BPs are directed acyclic layered edge-labeled multi-graphs. A graph is layered if the set of vertices $V = V_1 \mathbin{\dot\cup} V_2 \mathbin{\dot\cup} \cdots \mathbin{\dot\cup} V_k$ so that edges only exist between adjacent layers V_i and V_{i+1}. Further we set $V_1 = \{s\}$ and $V_k = \{t\}$. A BP gets input words from Σ^n. Labels are elements from the set $\{x_1, \ldots, x_n, \overline{x}_1, \ldots, \overline{x}_n\}$. A word w is accepted by some BP if there is a path from s to t so that the labels are *consistent* with the input. I.e. if $w_i = 1$ then no label $\overline{x}_i$ must be read and if $w_i = 0$ vice versa. A label is *inconsistent* with the input if it is not consistent. Since BPs are a computation model which receives words of constant length, we need to consider families of BPs as well. The set of languages which are accepted by some family of polynomial-size BPs is written in bold face: **BP**. By [Raz91] and [RA97], we know that $\mathbf{BP} = \mathbf{NL}/poly = \mathbf{UL}/poly$. Thus **BP** is closed under complement.

To restrict some BP's computational power, one can make constraints. A major one is the restriction of the width of a BP, i.e. the size of the largest layer. We call polynomial-size BPs of bounded (constant) width BWBP. An

important result is that **BWBP** = $\mathbf{NC^1}$ [Bar89]. As a by-product, we know, that width 5 is always sufficient in the bounded width case. Another restriction is planarity which gives us - in combination with boundedness - the classes **PBP** and **PBWBP**. By [Han08] and [BLMS98] we know that planar bounded-width BPs are related to $\mathbf{AC^0}$ and $\mathbf{ACC^0}$.

Counting classes have already been defined for Turing machines and circuits. By counting the number of paths from s to t being consistent with the input word, we get BP counting classes: #**BP**, #**BWBP**, #**PBP** and #**PBWBP**. We know that $\#\mathbf{BWBP} \subseteq \#\mathbf{NC^1}$ [CMTV98] but we do not know if the inclusion is strict. But then again, we know that GAP-$\mathbf{NC^1}$ functions are differences of #**BWBP** functions, so GAP-$\mathbf{NC^1}$ = GAP-**BWBP** [CMTV98].

3 Results

In this section, we define ZAP counting complexity and show how it embeds in the context of $\#\mathcal{C}$ and GAP-$\mathcal{C}$. We will also have a look at ZAP-$\mathbf{NC^1}$ in particular and BPs. The justification of our results is given in later sections.

In the context of $\#\mathcal{C}$ we spoke of proof trees. We now call these proof trees *positive*, since we consider the case that a word is accepted by a circuit. But now imagine a circuit C rejects some input. Let C' be the negation[2] of C. If a word is rejected, we have, as many proofs for rejection in C as for acceptance in C'. So, analogously we define negative proof trees which are sub-trees of the unfolded circuit which show that the input is rejected. Obviously, there are positive proof trees iff there are no negative ones.

Given a family of circuits C, by $\#acc_C^+ \colon \Sigma^* \to \mathbb{N}$ we denote the function which gives us the number of positive proof trees for some input. $\#acc_C^-$ is the function for the negative proof trees. Further let

$$\#acc_C := \#acc_C^+ - \#acc_C^-,$$

i.e. we subtract the number of negative proof trees from the number of positive proof trees, which gives the number of negative proof trees a negative sign.

The notion of negative proof trees allows us to define the ZAP operator which gives us new counting complexity classes.

Definition 1. *Let $\mathcal{C}$ be some circuit-based complexity class. Then* ZAP-$\mathcal{C}$ *is the set of all functions counting proof trees (positive and negative) in circuits in the limits of $\mathcal{C}$, i.e. all functions of the form $\#acc_C$ for some $C \in \mathcal{C}$.*

ZAP-$\mathcal{C}$ functions are always of the form $\Sigma^* \to \mathbb{Z} \setminus \{0\}$. In the $\#\mathcal{C}$ case it turned out that counting proof trees is equivalent to the computation arithmetic circuits perform. But this only holds within circuits in negation normal form because negation gates are not directly treatable. The absence of the notion of negative proof trees limits one to monotone circuits. In the ZAP case we can also arithmetize - and this not only in monotone circuits.

[2] Insert a negation gate after the output node and make this negation gate the new output.

Theorem 1. *For a circuit family C, the function $\#acc_C \in$ Zap-$\mathcal{C}$ can be calculated by an arithmetic circuit based on C resulting from the following interpretation of the gates:*

- Negation *inverts the sign of its input:* $x \mapsto -x$.
- *An* And *gate with inputs $x_1, \ldots, x_k$ gets assigned the function*

$$x_1, \ldots, x_k \mapsto \prod_{i=1}^{k} \max\{0, x_i\} + \sum_{i=1}^{k} \min\{0, x_i\}$$

- *An* Or *gate with inputs $x_1, \ldots, x_k$ gets assigned the function*

$$x_1, \ldots, x_k \mapsto \sum_{i=1}^{k} \max\{0, x_i\} - \prod_{i=1}^{k} \max\{0, -x_i\}$$

- *Input gates now hold values in $\{-1, 1\}$; in particular the Boolean 0 becomes -1 and 1 stays the same.*

As one can see, if an And gate receives only true/positive inputs, then the values are multiplied. If there are negative inputs, those are added. The case Or gates is symmetric.

This lemma gives us an analogue to the $\#\mathcal{C}$ correspondence between positive proof trees and arithmetic circuits. The next theorem provides an analogy to the fact that Gap-$\mathcal{C}$ functions are differences of $\#\mathcal{C}$ functions.

Theorem 2. *Let $\mathcal{C}$ be in $\{\mathbf{AC}^i, \mathbf{NC}^i | i \geq 0\}$. A function f is in* Zap-$\mathcal{C}$ *iff there are functions g and h in $\#\mathcal{C}$, so that $g(w) = 0 \Leftrightarrow h(w) \neq 0$ for all inputs w and $f = g - h$.*

Theorem 1 shows Zap-$\mathcal{C}$ to be a generalization of $\#\mathcal{C}$ and theorem 2 a restriction of Gap-$\mathcal{C}$. That means Zap-$\mathcal{C}$ lies between $\#\mathcal{C}$ and Gap-$\mathcal{C}$ and suggests that we have a natural definition at hand.

Next we consider $\mathbf{NC^1}$ and BPs. First we need to define what a Zap BP is; especially we need a counterpart to negative proof trees in circuits. This part of the paper is based on [Dor13]. Positive proof trees coincide with paths from s to t in BPs. If some input is rejected, there are negative proof trees in the circuit and no path in the BP from s to t being consistent with the input word. If there is no path, then source and target are separated, i.e. there is a cut. So, we discovered that counting cuts gives us exactly what we need. Formally, a cut is a partition of the vertices V in two sets V_s and V_t, so that $s \in V_s$, $t \in V_t$ and all edges between those sets go from V_s to V_t and are inconsistently labeled with respect to the input. $\#paths_B \colon \Sigma^* \to \mathbb{N}$ is the number of paths and $\#cuts_B \colon \Sigma^* \to \mathbb{N}$ the number of cuts some input generates.

Based on that we define Zap for BPs.

Definition 2. *Let $\mathcal{B}$ be a BP-based complexity class. Then* Zap-$\mathcal{B}$ *is the set of all functions counting paths and cuts of some input in a BP B which is in the bounds of $\mathcal{B}$, i.e. all functions of the form $\#acc_B := \#paths_B - \#cuts_B$.*

By this definition we get i.e. **BP** as well as **BWBP**, **PBP**, and **PBWBP**. Obviously such functions are of the desired form $\Sigma^* \to \mathbb{Z} \setminus \{0\}$ because a path exists iff no cut exists. Our goal is to find relations between Zap circuits and Zap BPs.

We find upper and lower bounds for $\mathbf{NC^1}$ in BPs, which are similar to the # case.

Theorem 3. Zap-**BWBP** $\subseteq$ Zap-$\mathbf{NC^1}$ $\subseteq$ Zap-**PBP**

The first inclusion holds because we can use the idea for counting paths by matrix multiplication to count cuts. For the second inclusion we have a procedure so transform an circuit into a BP. The idea is that And gates correspond to serial computation in BPs and Or gates to parallel computation. We then use the observation that this construction results in planar graphs. This is convenient, since paths and cuts are dual in taking the dual graph which makes each face to an vertex and draws edges between adjacent faces. Paths and cuts switch places in the dual graph. This shows us that cuts are really the counterpart to negative proof trees. This gives us the following:

Corollary 1. Zap-**PBP** *is closed under inversion*[3] *and* Zap-**PBWBP** *stays planar under inversion.*

If we apply the construction of theorem 3 on arbitrary BPs, then we see, that we can make a BP planar by admitting it quasi-polynomial size.

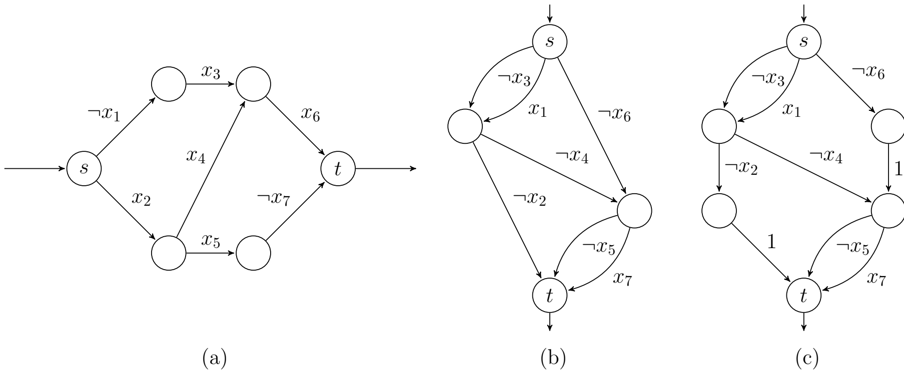


Fig. 1. Inversion in BPs. (a) is the original BP. (b) is the dual graph of (a), hence its inversion. (c) is the layered version of (b).

Figure 1 visualizes inversion in planar BPs. It remains a very interesting question how boundedness and planarity are related exactly and how to catch Zap-$\mathbf{NC^1}$ in terms of BPs.

[3] Inversion means multiplication with -1. That some circuit (or BP) is the inversion of some other circuit (BP) is a stronger requirement than that it is the negation.

4 Proofs

In this section we will provide the proofs for the theorems stated in the results section.

Proof (Theorem 1: ZAP *and arithmetic circuits).* We show the result by induction on the gates of the circuit. For input gates, we have one positive proof tree if the input is 1 and one negative proof tree if it is 0. Negation gates obviously realize an inversion of the sign. Now consider AND gates. Let g be some AND gate and assume that the inputs $x_1, \dots x_k$ already transport the right value. If the inputs are all true, g will output one and have a positive number of proof trees. In this case all input values have to be multiplied and this is what happens in this case. If one of the outputs is false, g must be assigned a negative number of proof trees by adding all negative inputs. The construction assures this. The OR case is shown analogously. □

The next Lemma allows us to split ZAP-$\mathcal{C}$ functions into a positive and a negative part. For a function f, we define $f^+ = \max\{f, 0\}$ and $f^- = \min\{f, 0\}$.

Lemma 1. *In the case of* $\mathcal{C} \in \{\mathbf{NC^i}, \mathbf{AC^i} | i \geq 0\}$*, it holds that if* $f \in$ ZAP-$\mathcal{C}$*, then* f^+ *and* $-f^-$ *are in* $\#\mathcal{C}$*.*

Proof. Let C be a family of circuits so that $\#acc_C = f$. We can assume that C is in negation normal form, since any negations can be pushed up to input level which does not change the arithmetic interpretation of the circuit. Calculating the number of positive proofs in a monotone circuit is exactly # counting as we know it, i.e. $f^+ \in \#\mathcal{C}$. The (positive) number of negative proofs is $-f^-$. In this case we use C and attach a negation gate after the output gate and push it to input level to get an equivalent negation normal form whose arithmetic interpretation counts negative proofs. So we get $-f^- \in \#\mathcal{C}$. □

The next result is the following lemma which states that circuits can be transformed in such a way that the realized ZAP function only takes absolute value 1. We use the regular definition for the sign function.

Definition 3. *The sign function sgn*: $\mathbb{Z} \to \{-1, 0, 1\}$ *maps negative values to* -1*, positive ones to* 1 *and* 0 *to itself.*

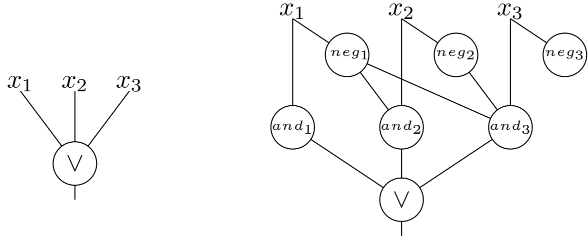
Now we prove that ZAP-$\mathcal{C}$ functions are closed under application of the sign function. This is the only way we know to reduce the absolute value. The lemma originates in [Fla12] and a related construction can be found in [Lan93].

Lemma 2. *If* f *is in* ZAP-$\mathcal{C}$*, then* $sgn \circ f$ *is also.*

Proof. Assume an arithmetic circuit C which computes f for some fixed input length. We inductively transform it into a circuit which computes $\text{sgn} \circ f$.
Input nodes by definition only take values in $\{-1, 1\}$. NEGATION gates leave values in this set. Now consider an OR gate v with inputs $x_1, \dots, x_k$ and assume

inductively all inputs to only take values in $\{-1, 1\}$. For each input we insert a NEGATION gate neg_i and an AND gate and_i. We make the following connections for all i (The construction is also pictured in following figure):

- x_i to neg_i
- x_i to and_i
- and_i to v
- neg_i to and_j for $j > i$



Construction of sgn in the case of OR with fan-in 3.

Now v only outputs values in $\{-1, 1\}$: Assume all x_i take value -1. Then all and_i output -1 and so does v. If there is at least one input which takes value 1, then let x_l be the one of those with smallest index l. In the construction only and_l will take the value 1 and hence v will output 1.
The construction for the case when v is an AND gate is easily adapted.
To stay in some complexity class, we need to note that this construction enlarges the circuit by a constant factor in depth as well as in size. □

An interesting interpretation of the previous lemma is that counting proof trees tells us nothing about efficiency of a circuit as one's intuition may suggest. In particular we made the circuit larger and the result is that we only get minimal numbers of proof trees: -1 and 1.

Next, we prove theorem 2, which gave us the relation to GAP-$\mathcal{C}$.

Proof (Theorem 2: ZAP-$\mathcal{C}$ functions as difference of #$\mathcal{C}$ functions). For a function $f \in$ ZAP-$\mathcal{C}$ we choose $g = f^+$ and $h = -f^-$. By Lemma 1, we know that $g, h \in \#\mathcal{C}$ and $g(x) = 0 \Leftrightarrow h(x) \neq 0$.

For the opposite direction, we are given g and h in #$\mathcal{C}$ so that $g(x) = 0 \Leftrightarrow h(x) \neq 0$. We construct a circuit whose arithmetic interpretation computes $g - h$. Let C_g and C_h be the corresponding monotonic Boolean circuits, whose arithmetization results in g respectively h. We construct a new Boolean circuit C in the way described in figure 2. The construction first adds the values of g and h using an OR gate. We show, that the construction using sgn and XOR ensures that the output's sign is inverted if $h(x) > 0$. Let ϕ be the sub-circuit $C_g \vee C_h$ (here and in the following we abuse notation by pretending we are dealing with formulas) and ψ be $sgn \circ h$. So the arithmetic interpretation of $\phi \oplus \psi$ gives us the end result. For the result we rewrite: $\phi \oplus \psi = (\phi \wedge \neg\psi) \vee (\neg\phi \wedge \psi)$. The first case is $g(x) > 0$. That means $(\phi \wedge \neg\psi)(x) = g(x)$ and $(\neg\phi \wedge \psi)(x) < 0$, hence the result is $g(x)$. The second case is $h(x) > 0$, so we get $(\phi \wedge \neg\psi)(x) = -1$ and

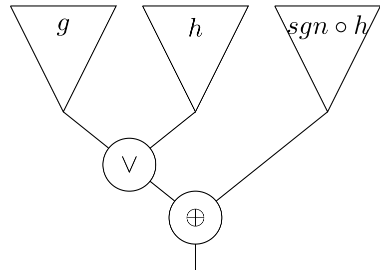
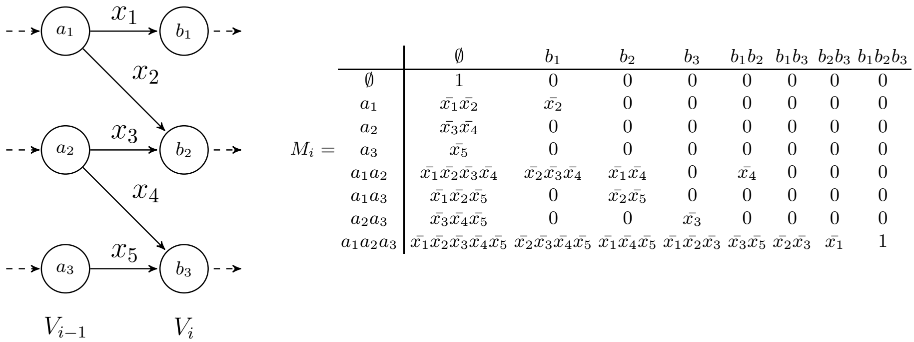


Fig. 2. Construction for the ZAP circuit computing $g - h$

$(\neg\phi \wedge \psi)(x) = -h(x)$. By applying the arithmetic semantic of the OR gate, we get $-h(x)$ as the desired result. Hence in arithmetic interpretation the circuit computes $g - h$. □

Proof (Theorem 3). We will only prove the first part of the theorem. (ZAP-**BWBP** $\subseteq$ ZAP-$\mathbf{NC^1}$). Let B be a BWBP. To prove the result, we have to show that $\#paths_B$ and $\#cuts_B$ are in $\#\mathbf{NC^1}$. By application of theorem 2, we have the desired result. $\#paths_B$ however is trivially in $\#\mathbf{NC^1}$ because that is how counting is defined in BPs and #BWBP $\subseteq \#\mathbf{NC^1}$.

We are left proving that counting cuts is possible in $\#\mathbf{NC^1}$. To do this we modify the construction for counting paths in $\#\mathbf{NC^1}$ which we will explain briefly. In the construction one defines matrices for each layer A_i. We can assume that the BP is not only layered but rasterized, i.e. layered also horizontally. If the BWBP is bounded by k then each node is addressable by the layer i and the position in the layer j for $1 \leq j \leq k$ as (i, j). Here we let $s = (1, 1)$ and $t = (p(n), 1)$. $p(n)$ is the polynomial bound for the number of layers. Now each matrix codes the edges between the layers. If we have a vector v_i which tells us in layer i how many paths are there from s to each node, then $v_i A_{i+1} = v_{i+1}$. It is $v_1 = (1, 0, \ldots, 0)$. Consider the matrix $A_i = (a_{pq})_{1 \leq p,q \leq k}$. We then have that a_{pq} is the sum of all labels for edges from $(i-1, p)$ to (i, q). The first element of the vector $v_1 A_2 \ldots A_{p(n)}$ is then the number of paths between s and t. Calculating this is possible in $\#\mathbf{NC^1}$, since multiplying two constant-sized matrices is possible in constant depth and if we multiply $p(n)$ many such matrices, we need a circuit if depth $\mathcal{O}(\log n)$, i.e. $\#cuts_B$ is calculable in $\#\mathbf{NC^1}$.



	$\emptyset$	b_1	b_2	b_3	b_1b_2	b_1b_3	b_2b_3	$b_1b_2b_3$
$\emptyset$	1	0	0	0	0	0	0	0
a_1	$\bar{x}_1\bar{x}_2$	$\bar{x}_2$	0	0	0	0	0	0
a_2	$\bar{x}_3\bar{x}_4$	0	0	0	0	0	0	0
a_3	$\bar{x}_5$	0	0	0	0	0	0	0
a_1a_2	$\bar{x}_1\bar{x}_2\bar{x}_3\bar{x}_4$	$\bar{x}_2\bar{x}_3\bar{x}_4$	$\bar{x}_1\bar{x}_4$	0	$\bar{x}_4$	0	0	0
a_1a_3	$\bar{x}_1\bar{x}_2\bar{x}_5$	0	$\bar{x}_2\bar{x}_5$	0	0	0	0	0
a_2a_3	$\bar{x}_3\bar{x}_4\bar{x}_5$	0	0	$\bar{x}_3$	0	0	0	0
$a_1a_2a_3$	$\bar{x}_1\bar{x}_2\bar{x}_3\bar{x}_4\bar{x}_5$	$\bar{x}_2\bar{x}_3\bar{x}_4\bar{x}_5$	$\bar{x}_1\bar{x}_4\bar{x}_5$	$\bar{x}_1\bar{x}_2\bar{x}_3$	$\bar{x}_3\bar{x}_5$	$\bar{x}_2\bar{x}_3$	$\bar{x}_1$	1

Fig. 3. Example for the construction of the cut-counting matrix M_i for layer i

Back to counting cuts: We do the same as in the path calculation, so that the first element of the vector $v_1 A_2 \ldots A_{p(n)}$ holds the desired value. Here we also calculate from layer to layer but this time we have matrices of size $2^k \times 2^k$. The reason is that we must regard the power-set of nodes in a layer. Recall that a cut is a partition of the vertices which separates s and t and so that the only edges between the two sets are inconsistent ones from V_s to V_t. An entry in a vector v_i corresponds to a set of nodes X in the layer i. The value is the number of cuts separating s from layers beyond i, so that $X \subseteq V_s$ and $\overline{X} \cap V_s = \emptyset$. By applying

$A_i = (a_{pq})_{1 \leq p,q \leq 2^k}$ we get this vector for the next layer. Say a_{pq} corresponds to sets X_1 and X_2. Then a_{pq} codes the possibility of extending a cut containing X_1 from the layer i to a cut containing X_2. If there are edges from $\overline{X}_1$ to X_2 then $a_{pq} = 0$ and else a_{pq} is the product of all negated labels of edges going from X_1 to $\overline{X}_2$. If there are no such edges at all then $a_{pq} = 1$. Figure 3 shows the construction.

We sketch the proof for the correctness of this construction based on a induction upon the layers of the BP. Let v_1 be the initial vector for the first layer which only includes s and let $v_i = v_{i-1}A_i$. Each component in such a vector stands for a subset X of nodes in a layer and we want to show that it holds the value how many cuts there are, so that exactly the nodes X in this layer belong to V_s. We assume inductively that this calculation is correct for layer i, i.e. $v_1 A_1 \dots A_i$ has the right values for each subset. Then $v_1 A_1 \dots A_i A_{i+1}$ is correct for layer $i+1$. Each subset Y in layer $i+1$ can become part of V_s if there is an extendable subset X from the previous layer. That means there are no edges going from V_t to V_s. The number of cuts with $Y \subseteq V_s$ is the sum of cuts of all subsets of the previous layer having no forbidden edges. Whether a forbidden edge occurs is dependent on the input; the matrix holds exactly the required constraints to the input so that a cut with some subset in layer i can be extended using exactly some subset in layer $i+1$. One can verify that this is exactly what happens of v_i is multiplied with A_{i+1}. □

5 Discussion and Future Work

In this work we introduced a meaningful counterpart to (positive) proofs trees in circuits, which are negative proof trees. Counting positive and negative proofs seems to be a natural thing to do. Based on some complexity class $\mathcal{C}$, Zap-$\mathcal{C}$ is the class of functions counting positive and negative proofs. We were also able to adapt our notion of negative proofs to branching programs. We were able to prove that Zap-**BWBP** $\subseteq$ Zap-$\mathbf{NC^1}$ $\subseteq$ Zap-**PBP**.

We think the notion of negative proofs is a neglected aspect of counting. We will outline possible applications.

The first one is the question whether #**BWBP** equals #$\mathbf{NC^1}$. By looking at Zap-$\mathbf{NC^1}$, we know that f is in Zap-$\mathbf{NC^1}$ iff f^+ and $-f^-$ are in #$\mathbf{NC^1}$. It seems rather unlikely that this also holds for BWBPs. If one proved that cuts in BWBPs cannot be counted in #**BWBP** then #**BWBP** would be separated from #$\mathbf{NC^1}$.

Another possible application is the well-known fact that **NL** is closed under complement. If we look at **NL**/*poly* which is equal to **BP** then we have the following implication: If Zap-**BP** is closed under inversion then **NL**/*poly* is closed under complement. Note that we could formulate such an implication also using #**BP**, but this seems less promising because of the absence of information concerning rejection of some input. As a first step one could try to re-prove the complement closure of **NL**/*poly* using our approach. Also **NL**/*poly* = **UL**/*poly* [RA97] could be a candidate to be re-proved. New proofs to those theorems would naturally give us new insights.

References

[All04] Allender, E.: Arithmetic circuits and counting complexity classes. In: Krajek, J. (ed.) In Complexity of Computations and Proofs. Quaderni di Matematica (2004)

[Bar89] Barrington, D.A.M.: Bounded-width polynomial-size branching programs recognize exactly those languages in NC^1. J. Comput. Syst. Sci. 38(1), 150–164 (1989)

[Bei93] Beigel, R.: The polynomial method in circuit complexity. In: Structure in Complexity Theory Conference, pp. 82–95. IEEE Computer Society (1993)

[BLMS98] Barrington, D.A.M., Lu, C.-J., Miltersen, P.B., Skyum, S.: Searching constant width mazes captures the AC^0 hierarchy. In: Meinel, C., Morvan, M., Krob, D. (eds.) STACS 1998. LNCS, vol. 1373, pp. 73–83. Springer, Heidelberg (1998)

[CMTV98] Caussinus, H., McKenzie, P., Thérien, D., Vollmer, H.: Nondeterministic NC^1 computation. J. Comput. Syst. Sci. 57(2), 200–212 (1998)

[Dor13] Dorzweiler, O.: Zap-Klassen für Schaltkreise und Branching Programs. Masterarbeit, Universität Tübingen (2013)

[FFK94] Fenner, S.A., Fortnow, L., Kurtz, S.A.: Gap-definable counting classes. J. Comput. Syst. Sci. 48(1), 116–148 (1994)

[Fla12] Flamm, T.: Zap-C Schaltkreise. Diplomarbeit, Universität Tübingen (2012)

[Han08] Hansen, K.A.: Constant width planar branching programs characterize ACC^0 in quasipolynomial size. In: IEEE Conference on Computational Complexity, pp. 92–99. IEEE Computer Society (2008)

[Jun85] Jung, H.: Depth efficient transformations of arithmetic into boolean circuits. In: Budach, L. (ed.) FCT 1985. LNCS, vol. 199, pp. 167–174. Springer, Heidelberg (1985)

[Lan93] Lange, K.-J.: Unambiguity of circuits. Theor. Comput. Sci. 107(1), 77–94 (1993)

[Pap94] Papadimitriou, C.H.: Computational complexity. Addison-Wesley (1994)

[RA97] Reinhardt, K., Allender, E.: Making nondeterminism unambiguous. In: FOCS, pp. 244–253. IEEE Computer Society (1997)

[Raz91] Razborov, A.A.: Lower bounds for deterministic and nondeterministic branching programs. In: Budach, L. (ed.) FCT 1991. LNCS, vol. 529, pp. 47–60. Springer, Heidelberg (1991)

[Ven92] Venkateswaran, H.: Circuit definitions of nondeterministic complexity classes. SIAM J. Comput. 21(4), 655–670 (1992)

[Vol99] Vollmer, H.: Introduction to circuit complexity - a uniform approach. Texts in theoretical computer science. Springer (1999)

[VT89] Venkateswaran, H., Tompa, M.: A new pebble game that characterizes parallel complexity classes. SIAM J. Comput. 18(3), 533–549 (1989)