THEORY OF COMPUTING, Volume 10 (12), 2014, pp. 297–339
www.theoryofcomputing.org

Width-Parameterized SAT: Time-Space Tradeoffs

Eric Allender* Shiteng Chen† Tiancheng Lou†
Periklis A. Papakonstantinou† Bangsheng Tang†

Received August 8, 2012; Revised July 14, 2013; Published October 8, 2014

Abstract: Alekhnovich and Razborov (2002) presented an algorithm that solves SAT on instances ϕ of size n and tree-width $\mathcal{TW}(\phi)$, using time and space bounded by $2^{O(\mathcal{TW}(\phi))}n^{O(1)}$. Although several follow-up works appeared over the last decade, the first open question of Alekhnovich and Razborov remained essentially unresolved: Can one check satisfiability of formulas with small tree-width in polynomial space and time as above? We essentially resolve this question, by (1) giving a polynomial space algorithm with a slightly worse run-time, (2) providing a complexity-theoretic characterization of bounded tree-width SAT, which strongly suggests that no polynomial-space algorithm can run significantly faster, and (3) presenting a spectrum of algorithms trading off time for space, between our PSPACE algorithm and the fastest known algorithm.

First, we give a simple algorithm that runs in polynomial space and achieves run-time $3^{\mathcal{TW}(\phi)\log n}n^{O(1)}$, which approaches the run-time of Alekhnovich and Razborov (2002), but

*Supported by National Science Foundation grants CCF-0832787 and CCF-1064785.

†Supported by National Basic Research Program of China Grant 2011CBA00300, 2011CBA00301, and the National Natural Science Foundation of China Grant 61033001, 61061130540, 61073174, 61250110577.

ACM Classification: F.1.3, F.2.2

AMS Classification: 68Q15, 68Q25

Key words and phrases: complexity theory, algorithms, complexity classes, circuit complexity, parameterized complexity, nondeterminism, CNF-DNF formulas, Boolean formulas, completeness, SAT, time-space tradeoff, treewidth, pathwidth

 DOI: 10.4086/toc.2014.v010a012

has an additional $\log n$ factor in the exponent. Then, we conjecture that this annoying $\log n$ factor is in general unavoidable.

Our negative results show our conjecture true if one believes a well-known complexity assumption, which is the $\mathsf{SC} \neq \mathsf{NC}$ conjecture and its scaled variants. Technically, we base our result on the following lemma. For arbitrary k, SAT of tree-width $\log^k n$ is complete for the class of problems computed by circuits of logarithmic depth, semi-unbounded fan-in and size $2^{O(\log^k n)}$ (SAC^1 when $k = 1$). Problems in this class can be solved simultaneously in time-space $(2^{O(\log^{k+1} n)}, O(\log^{k+1} n))$, and also in $(2^{O(\log^k n)}, 2^{O(\log^k n)})$. Then, we show that our conjecture (for SAT instances with poly-log tree-width) is equivalent to the question of whether the small-space simulation of semi-unbounded circuit classes can be sped up without incurring a large space penalty. This is a recasting of the conjecture that SAC^1 (and even its subclass NL) is not contained in SC.

Although we cannot hope for an improvement asymptotically in the exponent of time and space, we introduce a new algorithmic technique which trades constants in the exponents: for each ε with $0 < \varepsilon < 1$, we give an algorithm in time-space

$$\left(3^{1.441(1-\varepsilon)\mathcal{TW}(\phi)\log|\phi|}|\phi|^{O(1)},\ 2^{2\varepsilon\mathcal{TW}(\phi)}|\phi|^{O(1)}\right).$$

We systematically study the limitations of our technique for trading off time and space, and we show that our bounds are the best achievable using this technique.

1 Introduction

SAT is the prototypical NP-complete problem. In the real-world SAT instances tend to have structure. Also, in practice SAT-solvers abort due to lack of memory. In this paper we provide conclusive, asymptotically tight answers regarding time-space tradeoffs for SAT instances that are structured, where this structure is quantified by *tree-width*.[1]

This restriction of SAT was studied by Alekhnovitch and Razborov [2] (for references prior to this see within), who gave algorithms that work in time $2^{O(\mathcal{TW}(\phi))}|\phi|^{O(1)}$ and in space $2^{O(\mathcal{TW}(\phi))}|\phi|^{O(1)}$, where $\mathcal{TW}(\phi)$ is the tree-width of a CNF formula ϕ, and $|\phi| = n+m$ where n and m are the number of variables and clauses, respectively. The authors of [2] state their results in terms of the branch-width of the formula, which is within a constant factor of the tree-width. They conclude:

> "The first important problem is to overcome the main difficulty of the practical implementation which is the huge amount of space used by width-based algorithms. . . . Thus we ask if one can do anything intelligent in polynomial space to check satisfiability of formulas with small branch-width?"

The question raised by Alekhnovich and Razborov is a major issue in practical SAT-solving. Modulo complexity assumptions we fully answer this question.

[1] A formal definition of tree-width can be found in Section 2, an informal exposition is found earlier in Section 1.2.

1.1 Our contribution and techniques

We devise a simple space-efficient algorithm for SAT instances in CNF, which runs in time

$$3^{\mathcal{TW}(\phi)\log|\phi|}|\phi|^{O(1)}$$

and space

$$|\phi|^{O(1)}.$$

This is the first algorithm with running time exponential in the tree-width of the incidence graph (unlike [4] which is just on the primal graph) of arbitrary CNF instances (unlike [19] which is just for k-CNFs), that runs in polynomial space. Compared to the question of [2] this algorithm suffers a $\log|\phi|$ factor in the exponent of the running time. Our work revolves around this logarithmic factor. First, we conjecture that it cannot be removed:

Conjecture 1.1. *Let $\mathcal{A}$ be an algorithm for* SAT *that runs in time $2^{\mathcal{TW}(\phi)\delta(|\phi|)}|\phi|^{O(1)}$. Consider CNF formulas where $\mathcal{TW}(\phi) = O(|\phi|^{1-\varepsilon})$, for some fixed $\varepsilon < 1$. If $\delta(\phi) = o(\log|\phi|)$ then $\mathcal{A}$ uses space $2^{\Omega(\mathcal{TW}(\phi))}$.*

Second, we show that the above conjecture is equivalent to a widely-believed computational complexity assumption (scaled for a wider range of parameters). That is, we offer a complexity-theoretic connection that supports this conjecture. This computational complexity conjecture comes under the cryptic statement $\mathsf{NC} \not\subseteq \mathsf{SC}$. This is well-known to complexity theorists, and also of immense practical interest. SC is the class of problems that can be decided by algorithms that work simultaneously in poly-logarithmic space and polynomial time (i. e., efficient time and efficient space computation). NC is the class of problems that can be decided by circuits simultaneously of polynomial size and poly-logarithmic depth (i. e., parallel computation which uses a small number of processors and small parallel time). There is an, almost exact, correspondence between algorithm space and circuit depth (e. g., given an algorithm that uses poly-logarithmic space we can construct a parallel algorithm that runs in poly-logarithmic parallel time) and between algorithm time and circuit size. These correspondences are believed to break down when we *simultaneously* bound "time and space" and simultaneously bound "size and depth." That is, $\mathsf{NC} \not\subseteq \mathsf{SC}$ means that in general we cannot trade efficient #parallel processors-parallel time computation for efficient sequential time-space computation. Details and additional intuition are given in Section 3.

Semi-unbounded combinatorial circuits play an important role in this work. A semi-unbounded circuit (SAC) has AND gates of bounded fan-in, OR gates of unbounded fan-in, and all the negations at the input level. Despite their exotic nature these circuits have essential differences from bounded and unbounded fan-in circuits; see Section 3.1 for a discussion.

In Section 3 we show:

Theorem 3.5. SAT of a given tree decomposition of width $\log^k n$ is complete for the class

$$\mathsf{SAC}^k_{\mathsf{quasi}} := \mathsf{SAC}\big(O(\log n), 2^{O(\log^k n)}\big),$$

i. e., semi-unbounded fan-in circuits of $O(\log n)$-depth, and $2^{O(\log^k n)}$-size.

This is shown through a generic reduction in the spirit of [20]. We observe that NSC^k, defined as $\mathsf{NSC}^k = \mathsf{NTISP}(n^{O(1)}, O(\log^k n))$ is contained in $\mathsf{SAC}^k_{\mathsf{quasi}}$ which is in turn contained in NSC^{k+1}, where NTISP denotes the set of problems decidable by non-deterministic Turing Machines that are simultaneously TIme-SPace bounded, and we show:

Theorem 3.6. SAT of a given path decomposition of width $\log^k n$ is complete for NSC^k,

Note that the NSC levels are direct space-scaled analogs of NL and these SAC classes are direct size-scaled analogs of SAC^1. Therefore, separating the complexity of SAT parameterized by path-width and tree-width is equivalent to separating these classes, and hence, by padding, separating NL and $\mathsf{LOGCFL} = \mathsf{SAC}^1$. More importantly, putting these developments together we conclude that our conjecture implies $\mathsf{NC} \neq \mathsf{SC}$, and in fact as the tree-width ranges over different functions of the input length our conjecture is shown to be equivalent to a resource-scaled analog of $\mathsf{NC} \neq \mathsf{SC}$. Somewhat less rigorously:

> *Shrinking down the space, even by just a little, in a reasonably fast algorithm for* SAT *of bounded tree-width, is the same as saying that every highly parallelizable problem can be sequentially computed simultaneously in small time and space.*

Assuming for now that Conjecture 1.1 holds, it makes sense to devise algorithms that approach these limits. This is the topic of Section 4, which constitutes the more technically involved part of this work, though the practical significance of these algorithms is debatable.

We use our space-efficient algorithm, together with a time-efficient dynamic programming algorithm (essentially the algorithm of [35]), as the "end-points" for a spectrum of algorithms that trade off time and space complexity between these two extremes. But there is a catch. If we combine the time-efficient dynamic programming algorithm and our recursive algorithm in the obvious way, then we gain the worst of both worlds. Here "obvious" means that we discretize the space of truth assignments during the execution of the recursive algorithm and combine using dynamic programming. Instead, we introduce an implicit family of proof systems. We use two free parameters to specify an algorithm in this family. One parameter is an integer which is at least 2. This controls the "complexity" of the rules applied, for performing an unbalanced type of recursion of some sort. The larger this parameter is, the smaller the running time is and the more space is used. The second parameter is a real number in $(0,1)$ that controls the discretization of the truth assignment space. This family of algorithms is presented in Section 4, and in its full generality in Section 5. In the same sections we show that all of the infinitely-many pairs of values are of interest, depending on the different time-space bounds one may want to achieve.

Remark 1.2. Throughout this paper we assume that the tree (or path) decompositions are given in the input. To the best of our knowledge, the same is true in all other works in width-parameterized SAT.

1.2 Related work

Tree-width is a popular graph parameter introduced by Robertson and Seymour [32, 33]. The smaller the tree-width of a graph, the more the graph looks like a tree in some topological sense; for a connected graph of n vertices tree-width 1 means that the graph is a tree, whereas tree-width $n-1$ means that it is the complete graph. Several hard computational problems on general graphs become computationally easier when the input graph is of small tree-width; see, e. g., [7] for a survey.

For SAT instances the tree-width of a CNF formula is the tree-width of its associated graph (e. g., incidence graph, primal graph, or intersection graph). Among those graphs, the most general one is the incidence graph (a bipartite graph where one side has variable-nodes and the other clause-nodes). In some sense, the tree-width value on the incidence graph lower bounds the tree-width value of the rest [37]. In particular, the tree-width of the incidence graph of a CNF formula can be arbitrarily smaller than the tree-width of the CNF-formula graphs that were studied by Bacchus et al. [4]. There is a vast literature (too large to concisely cite here) in empirical and theoretical studies in various width-parameterizations of SAT—we only cite some of the most relevant ones below.

Algorithms for width-parameterized SAT**.** Prior to our work, [4, 19] addressed the question of Alekhnovich and Razborov. In [19] the authors gave a combinatorially non-explicit algorithm only for the k-SAT problem, where k is constant, where the algorithm runs in time $2^{O(\mathcal{TW}(\phi)\log|\phi|)}$ and space $|\phi|^{O(1)}$, when $\mathcal{TW}(\phi) = \Omega(\log|\phi|)$. The deficiencies of [19] (which we overcome in our current paper) are not only that their algorithm works only for k-SAT, but also that the constant in the exponent of the running time cannot be bounded in any easy way due to the non-explicitness of the argument presented there. Bacchus et al. [4] present a polynomial-space DPLL algorithm with running time exponential in the tree-width of the *primal graph* of a formula, hence their SAT algorithm is strictly weaker than ours (although they also present algorithms for #SAT and similar problems).

There have been a number of follow-ups improving the running time of the Alekhnovich and Razborov algorithm [2], considering different width-parameters: Fischer et al. [16] give algorithms for SAT (and a somewhat generalized version of #SAT) parameterized by tree-width and clique-width. Their tree-width algorithm matches the running time and space of an algorithm of Samer and Szeider [35], which we make use of later in this paper as a time-efficient algorithm, running in time-space

$$\left(2^{2\mathcal{TW}(\phi)}|\phi|^{O(1)},\ 2^{\mathcal{TW}(\phi)}|\phi|^{O(1)}\right).$$

Also, we remark that algorithms (e. g., for graph problems) which replace the tree-width parameter $\mathcal{TW}$ in the exponent by a $\mathcal{TW}^2$ and at the same time reducing the space to polynomial (see e. g., [26]) are strictly worse than our algorithms (and in particular fail to reach our target lower bound for the Alekhnovitch-Razborov question). In particular, unlike the classical parameterized complexity approach, the interesting part of our treatment (and in particular our complexity results) are for values of tree-width that are related to the input length, and in fact $\Omega(\log n)$.

Improving constants, previous work, and what's different. Improving the constant in the base of an exponential time algorithm is a well-established goal in the field of exact computation for NP-hard problems; see e. g., [18, 40] for an overview. In particular, for k-SAT there is a line of work in algorithms that run in time α^n for $\alpha < 2$; e. g., [28, 31, 36, 40]. An issue somewhat superficially related to our conjecture deals with time-space tradeoffs for algorithms for NP-hard permutation problems, as discussed for example in [9] and the references within (in particular [24]). However, there is no easy way to adapt these techniques in our setting, and if Conjecture 1.1 is true, they cannot really be applied at all. A key property of these previous algorithms is that as smaller subproblems are considered, the parameter *number of nodes* becomes smaller. There is no obvious way to achieve this when the parameter is the *width* of the formula.

Finally, to the best of our knowledge our work is the first to address the issue of smooth time-space tradeoffs for width-parameterized SAT. Prior to our work there are others which solely discuss lower bounds on the running time; e. g., for graph problems (and under the very strong ETH assumption [22]) [25]. Another kind of tradeoff (between size of the separator and the sharpness) for graph problems was discussed in [17]. We should also mention that in two excellent works on constraint satisfaction problems Grohe [21] (assuming that $\mathsf{FPT} \neq \mathsf{W}[1]$) and Marx [27] (assuming ETH) essentially show that the running time of the known width-based algorithms is optimal.

Hardness results and complexity characterizations. Every problem in NP can be seen as a problem where accepting instances can be verified in logarithmic space (i. e., we can settle for less than polynomial time in the verification). SAT of bounded path-width has been shown [30] complete for the class of problems that can be decided by a logarithmic space machine which has "streaming access" to the tape containing the witness; i. e., scanning the witness tape at most a given number of times. In particular, $O(r(|\phi|))$ passes correspond to SAT instances with given path-decompositions of width $r(|\phi|)\log|\phi|$. Specifically, deciding formulas with given path decompositions of width $O(\log|\phi|)$ is complete for NL, and [30] asks whether the complexity of SAT instances when the parameter is tree-width $O(\log|\phi|)$ is more difficult. In this paper we answer this question affirmatively, unless[2] $\mathsf{NL} = \mathsf{SAC}^1$. Furthermore, we show an exact correspondence of these "streaming verification" classes with the levels of the known NSC hierarchy. Our new characterization through semi-unbounded circuits is of independent interest, and seems more natural than the characterizations presented in [1].

Relation to Propositional Proof Complexity. Our work opens new, exciting directions for Propositional Proof Complexity. One way to make progress towards validating Conjecture 1.1 is to restrict attention to specific types of algorithms. The study of restricted proof systems is one such choice. In fact, Beame, Beck, and Impagliazzo [6] very recently made progress towards *exactly* validating our question. In particular, they proved a Resolution Refutation size-space tradeoff, which implies that there exists a family of formulas ϕ of tree-width $\mathcal{TW}(\phi)$ where for every $k > 0$ every resolution refutation of size n^k requires space

$$2^{\mathcal{TW}(\phi)\frac{\log\log n}{\log\log\log n}}.$$

This very significant development is the first super-polynomial lower bound of this sort, and through our work it can be interpreted as validating the $\mathsf{SAC}^1 \not\subseteq \mathsf{SC}$ conjecture, at least for a class of restricted algorithms. This is a new direction; lower bounds in proof complexity are clearly connected to the $\mathsf{NP} \neq \mathsf{coNP}$ conjecture [15], but have not previously seemed to have a bearing on the $\mathsf{SC} \neq \mathsf{NC}$ question.

2 Preliminaries

We introduce notation, terminology, and conventions used throughout the paper.

[2]It is conjectured, e. g. [14], that $\mathsf{SAC}^1 \neq \mathsf{NL}$. Note that SAC^1 is also known as LOGCFL.

2.1 Notation

All logarithms are of base 2, and all propositional formulas are in Conjunctive Normal Form (CNF). SAT is the decision problem where given an arbitrary CNF formula we want to decide if it is satisfiable. We let k-SAT denote the restriction of SAT to CNFs where each clause has at most k literals. For a formula ϕ, m denotes the number of clauses, n the number of variables, and C_i and x_j stand for the i-th clause and j-th variable respectively. For convenience we write $|\phi| = m+n$. When there is no confusion (e. g., when defining complexity classes) n is used to denote the input length.

2.2 Tree-width

Definition 2.1. Let $G = (V,E)$ be an undirected graph. A *tree decomposition* of G is a tuple $(\mathcal{T},X)$, where $\mathcal{T} = (W,F)$ is a tree, and $X = \{X_1,\cdots,X_{|W|}\}$ where $X_i \subseteq V$ such that

(1) $\bigcup_{i=1}^{|W|} X_i = V$,

(2) $\forall (i,j) \in E$, $\exists t \in W$, such that $i,j \in X_t$,

(3) $\forall v$, the set $\{t : v \in X_t\}$ forms a subtree of $\mathcal{T}$.

Each of X_i is called a *bag*, the width of (T,X) is defined as $\max_{t\in W} |X_t| - 1$, and the *tree-width* $\mathcal{TW}(G)$ of graph G is defined as the minimum width over all possible tree decompositions.

When the tree decomposition $\mathcal{T} = (W,F)$ is restricted to a path, the decomposition is called a *path decomposition*, and the specific tree-width is called the path-width $\mathcal{PW}(G)$. The following inequality holds (e. g., [8])

$$\mathcal{TW}(G) \leq \mathcal{PW}(G) \leq O\left(\log |V| \cdot \mathcal{TW}(G)\right). \tag{2.1}$$

Definition 2.2. The *incidence graph* G_ϕ of a SAT instance ϕ is a bipartite graph, where in one side of the bipartization each node is associated with a distinct variable, and in the other each node is associated with a clause. There is an edge between a clause-node and a variable-node if and only if the variable appears in a literal of the clause. The tree-width of a formula ϕ is the tree-width of its incidence graph, $\mathcal{TW}(\phi) = \mathcal{TW}(G_\phi)$. *When it is clear from the context we may abuse notation and write* $\mathcal{TW}(\phi)$ *to denote the width of a given decomposition of* G_ϕ.

We assume that a tree decomposition of the incidence graph of ϕ is given as input along with ϕ. For convenience, we assume without loss of generality that the input tree decompositions $((W,F),X)$ have the following two properties.

(1) $|W| = O(\mathcal{TW}(\phi) \cdot |V|) = O(\mathcal{TW}(\phi)|\phi|)$, and

(2) the tree $\mathcal{T} = (W,F)$ has bounded degree 3.

We call a tree decomposition *nice* if it satisfies the two properties above. A tree decomposition can be converted to a nice one in linear time (see e. g., [23, 8]). The maximal degree in the tree decomposition is denoted by d. By the property above, $d \leq 3$. If the input is given with a *path* decomposition, then $d \leq 2$.

Remark 2.3. We present our results with the parameter d. One may replace d by 2 or 3 when the structure of the input decomposition is known to be a path or a (nice) tree.

2.3 Assignments

We introduce terminology and notation to talk about truth assignments on bags. Let X be a bag in the tree decomposition, $\mathcal{V}$ be the variables and $\mathcal{C}$ be the clauses in X. Also, $n_{\mathcal{V}} = |\mathcal{V}|$ and $m_{\mathcal{C}} = |\mathcal{C}|$. An *assignment* R_X for X is a binary vector of length $n_{\mathcal{V}} + m_{\mathcal{C}}$. The first n_V bits indicate the truth values of the corresponding variables. Note that the term "assignment" does not correspond only to a "truth assignment" on the variables in X. It is an assignment of bit values both to variables and to clauses.

What values the last $m_{\mathcal{C}}$ bits have is a subtle issue explained in Section 4. For the dynamic programming algorithm things are pretty clear. However, for the space-efficient and trade-off algorithms, things become more subtle. Intuitively, a bit corresponding to a clause C is 1 if we "have decided" to eventually satisfy this clause (this has to do with where we are in the execution of the algorithm). Such a decision is different for different algorithms, but we use the same data-structure.

Actually, the most straightforward way of defining the clause bits is to let it denote whether the corresponding clause "is" satisfied. To ensure that a clause is satisfied in one of the branches in the tree decomposition, we need to enumerate all $2^d - 1$ combinations of branches on which the clause is satisfied. However, if one is interested in only the satisfiability problem (and not, e. g., in #SAT) we observe that d combinations suffice.

3 A complexity-theoretic characterization

We show that (i) our Conjecture 1.1 is equivalent to a widely believed complexity assumption (and its scaled analogs), and (ii) under a different well-known complexity assumption ($\mathsf{NL} \subsetneq \mathsf{LOGCFL}$), for the same width parameter $w(|\phi|)$ SAT of tree-width $O(w(|\phi|))$ cannot be efficiently reduced to SAT of path-width $O(w(|\phi|))$. Both of these results follow by first proving that SAT parameterized by path- and tree-width is complete for natural complexity classes. To obtain these results, we heavily rely on properties of semi-unbounded combinatorial circuits. In Section 3.1 we give a primer with basic intuition about these circuits. Also, based on properties of these circuits and on the relation between path-width and tree-width in equation (3.1) in Section 3.3, we provide a new characterization of NSC.

3.1 A primer on semi-unbounded fan-in circuits

The statements of the lower bounds do not explicitly refer to semi-unbounded families of circuits, but we use them inside the arguments. A circuit is *semi-unbounded* when it has unbounded fan-in OR gates, bounded fan-in AND gates, and all the negations are at the input level. The class of problems that can be decided by such circuits of depth $O(\log^i n)$ and polynomial size is denoted by SAC^i. Clearly, $\mathsf{NC}^i \subseteq \mathsf{SAC}^i \subseteq \mathsf{AC}^i$, where NC and AC denote the complexity classes characterized by the same parameters with bounded fan-in and unbounded fan-in families of circuits, respectively. There is also the issue of uniformity; we provide the necessary background regarding circuit uniformity in the next sub-section. For the moment, the reader should make use of an informal uniformity assumption, meaning merely that there is an efficient algorithm describing how to construct the circuits for inputs of size n.

What is special in SAC circuits? Suppose that you are given oracle access to the description of an *unbounded* circuit C (with all negation gates on the input level) and an input x, and you want to verify recursively that $C(x) = 1$. The standard algorithm would be to start at the output gate and execute the following recursive algorithm: if the current gate g is an OR gate, nondeterministically pick a gate h that feeds into g and verify that h evaluates to 1; if the current gate g is an AND gate with $h_1, \ldots, h_m$ feeding into it, recursively verify that each of the h_i evaluates to 1. An accepting run of this algorithm corresponds to a (possibly huge) tree, called a "proof tree" [38]; if the circuit C has bounded fan-in (or even semi-unbounded fan-in), then the size of this tree is bounded by 2^{depth}. In the bounded fan-in case, restricting the depth of C automatically implies restricting the size of C; in the semi-unbounded fan-in case this is not true. The important aspect of semi-unbounded fan-in circuits that we will utilize, is that the proof tree can be much smaller than the circuit (unlike the bounded-fan-in case), and has size exponential in the depth (unlike the unbounded-fan-in case).

An observation on uniformity. Consider a family of semi-unbounded circuits of size $2^{n^{O(1)}}$ and depth $O(\log n)$ that is polynomial time uniform, in the sense that, given the names of two gates g and h, one can determine in time polynomial in n whether there is an edge from g to h, and what kind of gates g and h are. (Note that $n^{O(1)}$ bits are required, merely to write down the name of one of the gates.) Then, for an input x, $|x| = n$, the problem of evaluating the membership of the family $C_n(x)$ is in NP. That is, given x we guess a proof as described above and then verify that it is a valid proof. Clearly, every problem $L \in \mathsf{NP}$ can be computed by such a circuit since in $\log n$ depth we can verify that a given witness y is an encoding of a valid accepting computation path on input x, and hence we obtain a circuit for $L \cap \{0,1\}^n$ by putting a big OR gate as the output gate, computing the disjunction, over all potential witnesses y, of the $O(\log n)$ depth circuit testing if y is a witness for the input x). In other words, NP is precisely the class of problems computed by uniform semi-unbounded circuits of size $2^{n^{O(1)}}$. Observe that if we do not insist on the uniformity, then an arbitrary function can be computed by a $2^{O(n)}$-size and $O(\log n)$-depth semi-unbounded (non-uniform) circuit.

Relations of SAC circuits to other models of computation. Semi-unbounded fan-in circuits are intimately related to Alternating Turing Machines (ATMs) and to Nondeterministic Auxiliary PushDown Automata ($\mathsf{NAuxPDA}$s). We provide definitions later on; for a more detailed treatment see, e. g., [34]. For the moment let us say that an ATM and a $\mathsf{NAuxPDA}$ are basically the same thing. Also, recall that an ATM is a nondeterministic Turing Machine with two kinds of nondeterministic states: existential and universal. An existential state is accepting if and only if at least one successor configuration is accepting, whereas a universal state is accepting if and only if each successor configuration is accepting. Although it requires a bit of work to show equivalence [38] it should come at no surprise that proofs for SAC circuits are related to ATM computations.

3.2 Complexity theory notation and some preliminaries

NSC is the nondeterministic analog of SC, the class of languages decidable simultaneously in polynomial time and poly-logarithmic space. Define $\mathsf{NSC}^k := \mathsf{NTISP}\big(n^{O(1)}, O(\log^k n)\big)$. It is widely conjectured that $\mathsf{SC} \neq \mathsf{NC}$. Here is a stronger intuitive form of this conjecture.

Conjecture 3.1. *The* NL*-complete graph reachability problem*[3] *cannot* simultaneously *be solved deterministically in sub-polynomial space and polynomial time. That is, depth-first search cannot be simulated quickly in small space, and hence* $\mathsf{NL} \not\subseteq \mathsf{TISP}(n^{O(1)}, n^{o(1)})$. *This implies the weaker conjecture* $\mathsf{SAC}^1 \not\subseteq \mathsf{TISP}(n^{O(1)}, n^{o(1)})$.

We denote by $\mathrm{SAT}_{\mathsf{tw}}(w(|\phi|))$ the problem of deciding SAT of a given CNF formula together with a tree decomposition of width $w(|\phi|)$. Similarly, for path-width we use the notation $\mathrm{SAT}_{\mathsf{pw}}(w(|\phi|))$. [30] shows that $\mathrm{SAT}_{\mathsf{pw}}(w(|\phi|))$ is complete for the class $\mathsf{NL}[w(|\phi|)/\log|\phi|]$, characterized by log-space bounded Turing Machines augmented with a polynomially long read-only, nondeterministic tape on which they make $O(w(|\phi|)/\log|\phi|)$ passes.

We use the notation $\mathsf{SAC}(\mathsf{depth}, \mathsf{size})$. We follow standard conventions when defining levels of the NC hierarchy, by defining $\mathsf{SAC}^k := \mathsf{SAC}(O(\log^k n), n^{O(1)})$. However, in this paper a different parameterization will be of equal importance: by restricting the depth of semi-unbounded fan-in circuits to be $O(\log n)$, and allowing the size to be quasipolynomial, we obtain subclasses of NP denoted by

$$\mathsf{SAC}^k_{\mathsf{quasi}} := \mathsf{SAC}\big(O(\log n), 2^{O(\log^k n)}\big)\,.$$

The study of SAC circuits, and the various classes SAC^i has received considerable attention, e. g., [12, 38]. The $\mathsf{SAC}^k_{\mathsf{quasi}}$ classes (very shallow quasi-polynomial size circuits) are introduced in this paper; they characterize the NSC hierarchy (equation (3.1)). For these families of circuits we use DLogTime-uniformity [5]. This means that the *direct connection language* for the circuit family can be recognized in linear time. The direct connection language takes inputs of the form $\langle n,i,d,j,t\rangle$ such that $d>0$ and the dth input of the gate i in the circuit for inputs of length n is of type $t (\in \{\mathsf{AND}, \mathsf{OR}, 0, 1\})$ and has index j, or else $d=0$ and gate i is of type t. Since the string $\langle n,i,d,j,t\rangle$ has length logarithmic in the size of the circuit for inputs of length n, it follows that, for $\mathsf{SAC}^k_{\mathsf{quasi}}$ circuits, questions about connectivity in the circuits for length n can be answered in time $O(\log^k n)$.

Simultaneously depth-size bounded semi-unbounded circuits are intimately related to space-time bounded NAuxPDAs. A NAuxPDA is a nondeterministic space-bounded Turing Machine equipped with an unbounded stack (see [13] for a precise definition). $\mathsf{NAuxPDA}(s(n), t(n))$ is the class of languages decidable by a NAuxPDA in space $O(s(n))$ and time $O(t(n))$. Although general Turing machine *time* is related to circuit *size* while circuit *depth* is related to *space*, on NAuxPDAs the correspondence is reversed; simultaneous bounds on circuit size and depth correspond to bounds on space and time, respectively. Generalizing the arguments in [34] and [38] we obtain:

Lemma 3.2. $\mathsf{SAC}^k_{quasi} = \mathsf{NAuxPDA}\big(O(\log^k n), n^{O(1)}\big)$*, for* $O(\log^k n)$ *time uniform* SAC *circuits.*

Proof. The proof of $\mathsf{SAC}^k_{\mathsf{quasi}} \supseteq \mathsf{NAuxPDA}(O(\log^k n), n^{O(1)})$ can be shown by following the proof for the special case $k=1$ (i. e., $\mathsf{NAuxPDA}(O(\log n), n^{O(1)}) = \mathsf{SAC}(O(\log n), n^{O(1)})$) [34, 38]. (See also [39].) However, in the proof of Lemma 3.8 we will need to assume that the uniform $\mathsf{SAC}^k_{\mathsf{quasi}}$ have certain properties, and thus we follow a different outline here, to establish that those properties hold.

Ruzzo [34, Theorem 1 & Corollary 7] showed that any language in $\mathsf{NAuxPDA}(O(\log^k n), n^{O(1)})$ is accepted by a NAuxPDA respecting these same resource bounds, where additionally the height of the

[3] Given a directed graph $G=(V,E)$ and two designated vertices $s,t\in V$, is t reachable from s?

pushdown is $O(\log^{k+1} n)$ (and pushes and pops consist of moving strings of length $O(\log^k n)$ to and from the stack—hence it is useful to think of the stack as having height $\log n$, over "symbols" of length $O(\log^k n)$). It is easy to see that such a machine can also be assumed to be somewhat "oblivious," in the sense that the positions of the worktape and input heads at time t are the same for all inputs of length n. Rossmanith and Niedermeier subsequently improved on this, to show that the NAuxPDA can be assumed to be completely oblivious, in the sense that the sequences of pushes and pops are also the same for all inputs of length n [29, Theorem 28]. (Rossmanith and Niedermeier state their theorems in terms of machines with a logarithmic worktape bound, but their proof works also for larger space bounds, as long as the time is polynomial.) In particular, the pushes and pops follow a very regular pattern, so that the computation is divided into phases corresponding to the height of the stack. The computation starts and ends with stack height zero, and precisely half-way through the computation, the stack height is also zero. Call these three configurations C_0, D_0, and E_0. The computation from C_0 to D_0 and from D_0 to E_0 all takes place with a stack height of at least 1 "symbol" (where the stack "symbols" are of $O(\log^k n)$ bits each); these are the two "phases" with height 1. In general, there are 2^i phases with height i, for each $i \le i_{\max} = O(\log n)$. Each such phase (for $i < i_{\max}$) has some start configuration C_i and end configuration E_i that take place at times that depend only on the input length n, and there is a configuration D_i that also has stack height i, such that the computations between C_i and D_i and between D_i and E_i have exactly the same length and are both phases with stack height $i+1$. (The phases at height $i_{\max}$ start in a configuration C that has a number $j \le n$ recorded in it, and ends in a configuration that records the j-th input symbol; no stack manipulation occurs in such a phase.)

Thus in order to show that

$$\mathsf{NAuxPDA}\big(O(\log^k n), n^{O(1)}\big) \subseteq \mathsf{SAC}^k_{\mathsf{quasi}},$$

it suffices to build circuits to simulate oblivious machines that have this very restrictive computation pattern. The output gate will check if the height zero phase starts with the initial configuration C_0 and ends with the accepting configuration E_0; it is an OR gate, connected to gates labeled with triples (C_0, D_0, E_0) for all D_0, to see if there is a computation from C_0 to E_0 passing through D_0. In general, gates labeled (C_i, D_i, E_i) (or (C_i, D_i, E_i, γ)) where C_i, D_i, and E_i encode the worktape contents and input head positions (but not the stack contents) for some phase with stack height i (and γ is a stack symbol of length $\log^k n$) are AND gates, testing whether there are computations from C_i to D_i and from D_i to E_i, respectively. The children of these AND gates, corresponding to some computation between stack height i configurations A and B, are OR gates over all $(C_{i+1}, D_{i+1}, E_{i+1}, \gamma)$ such that:

- there is a move from A to C_{i+1} pushing γ, and
- there is a move from E_{i+1} to B popping γ.

(If $i+1 = i_{\max}$, then instead of $(C_{i+1}, D_{i+1}, E_{i+1}, \gamma)$, the gates have the format $(C_{i+1}, E_{i+1}, \gamma)$, and these gates are (possibly negated) input gates, recording whether the given input symbol is consistent with a transition from C_{i+1} to E_{i+1}.)

It should be clear that the circuit directly simulates the NAuxPDA. For more details about the uniformity of the circuits, we refer the reader to [29, 38, 39].

Now we prove the other direction:

$$\mathsf{NAuxPDA}\big(O(\log^k n), n^{O(1)}\big) \supseteq \mathsf{SAC}\big(O(\log n), 2^{O(\log^k n)}\big).$$

Let L have $\mathsf{SAC}(O(\log n), 2^{O(\log^k n)})$ circuits. A $\mathsf{NAuxPDA}$ accepts L as follows. On input x, compute the name of the output gate of the circuit (call it g), and write g on the worktape. Start the routine EVAL(g), described below.

Algorithm 1 EVAL(g)

if g is a (negated) input gate connected to input bit x_i **then**
    accept iff x_i is 1 (0, respectively)
end if
if g is an OR gate **then**
    nondeterministically guess a gate name h and check that $h \to g$ is an edge in the circuit
    return EVAL(h)
end if
if g is an AND gate **then**
    compute the gates h_1 and h_2 that feed into g
    push h_2 onto the stack and call EVAL(h_1)
    if this evaluates to 0 **then**
        halt and reject
    else
        return EVAL(h_2)
    end if
end if

The run-time required to to evaluate a gate g at depth d is $2^d n^{O(1)}$, assuming $\mathsf{LOGSPACE}$ uniformity. This is polynomial in n, since $d = O(\log n)$. The space required is dominated by the number of bits needed, to write down the name of a gate, which is $O(\log^k n)$.

This completes the proof, but let us mention here that later, in Lemma 3.7 and Lemma 3.8, we show that $\mathrm{SAT}_{\mathsf{tw}}(\log^k |\phi|)$ is hard for $\mathsf{SAC}(O(\log n), 2^{\log^k n})$, and is contained in

$$\mathsf{NAuxPDA}\big(O(\log^k n), n^{O(1)}\big). \qquad \square$$

The reader may be surprised that acceptance of a super-polynomial size circuit can be verified in (nondeterministic) polynomial time. This is related to the structure and size of *proofs* of accepting inputs for semi-unbounded circuits. In particular, the size of such a proof/certificate is exponential in the depth of the circuit (see the proof of Lemma 3.8 for details).

3.3 Completeness for $\mathrm{SAT}_{\mathsf{pw}}(\log^k |\phi|)$ and $\mathrm{SAT}_{\mathsf{tw}}(\log^k |\phi|)$, and a new circuit characterization of the NSC hierarchy

In Theorem 3.5 we show that $\mathrm{SAT}_{\mathsf{pw}}(\log^k |\phi|)$ is complete for NSC^k and Theorem 3.6 states that $\mathrm{SAT}_{\mathsf{tw}}(\log^k |\phi|)$ is complete for $\mathsf{SAC}^k_{\mathsf{quasi}}$. We remark that the tree-width/path-width relation $\mathcal{PW}(G) \leq$

$\mathcal{TW}(G)\log n$ can be shown via a reduction computable in $\mathsf{LOGSPACE}$ [10]. Putting these together (or this can also be seen via a direct argument) we have the following characterization of the NSC levels:

$$\underbrace{\mathsf{NL}}_{\mathsf{NSC}^1} \subseteq \underbrace{\mathsf{SAC}^1}_{\mathsf{SAC}^1_{\mathsf{quasi}}} \subseteq \mathsf{NSC}^2 \subseteq \mathsf{SAC}^2_{\mathsf{quasi}} \subseteq \mathsf{NSC}^3 \subseteq \cdots \subseteq \mathsf{NSC} = \mathsf{SAC}_{\mathsf{quasi}}\,. \tag{3.1}$$

Our completeness results require us to present upper bounds on the complexity of SAT with small tree-width and path-width. For these upper bounds, we need the notation of *consistency*. Since we have extended the notion of assignment to also include assignments to *clauses*, we also need to have a correspondingly extended notation of consistency of assignments. The rigorous definition of consistency is deferred until the next section; for this section it suffices to rely on an intuitive understanding of the notion. Intuitively, assignments to two bags are said to be consistent, if the bits corresponding to variables agree, and some additional constraints imposed by the bits corresponding to clauses are satisfied such that a satisfying truth assignment can be deduced. For this section, it suffices to know that, if assignments for two bags are written on the worktape, then it is very easy to determine if the assignments are consistent. Also, by the connectivity properties of tree decompositions, it suffices to check consistency of neighboring bags.

Now, we turn to showing these completeness results. The following lemma implies Theorem 3.5.

Lemma 3.3. $\mathsf{NSC}^k = \mathsf{NL}[\log^{k-1} n]$, *for* $k \in \mathbb{Z}^+$.

Proof. Let's see why $\mathsf{NSC}^k \subseteq \mathsf{NL}[\log^{k-1} n]$ first. Let M be a machine that accepts a language $L \in \mathsf{NSC}^k$. From M, we construct a machine M' that uses only logarithmic space on its worktape, and that makes $O(\log^{k-1} n)$ passes over a tape of polynomial length that holds the sequence of "nondeterministic" bits. On accepting computations, the nondeterministic tape of M' will contain an encoding of a computation of M: i. e., a sequence of encodings of successive configurations (from initial state to accepting state) of a complete run of M accepting the given input. (Clearly, such an encoding will have polynomial length since the running time of M is polynomial and the length of each configuration is $O(\log^k n)$.) A configuration will include state, head position and worktape. Without loss of generality we assume that all the encodings of configurations have the same size, and that the worktape is divided evenly into blocks of length $O(\log n)$. Note that because of the locality of computation, two adjacent configurations only differ in $O(1)$ bits; the ith blocks of the worktape of two consecutive configurations will be identical when the head is not in the corresponding block, and otherwise will differ only in $O(1)$ bits.

In the ith pass, starting from the initial configuration, M' will check that the ith blocks of each two consecutive configurations are correct. To do this, M' will read blocks $i-1, i$, and $i+1$ of each two consecutive configurations into its worktape in turn, as well as the state and head position of both configurations. If the head is not in the ith block, then M' will merely check that ith blocks of the two configurations are identical; if the head is in the ith block, M' will check whether the move is a legal move of M. Some additional bookkeeping is necessary when the head is moving into or out of the ith block; in those cases, the blocks $i-1$ and $i+1$ will also need to be consulted. If the ith blocks of configurations j and $j+1$ are deemed to be consistent, then the process is repeated for configurations $j+1$ and $j+2$. It should be clear that M' uses logarithmic space and makes only $O(\log^{k-1} n)$ passes over its nondeterministic tape.

For the other direction, it is sufficient to present a complete problem for $\mathsf{NL}[\log^{k-1} n]$ that is contained in NSC^k. $\mathrm{SAT}_{\mathsf{pw}}(\log^k |\phi|)$ is such a problem, by the following characterization:

Lemma 3.4 ([30])**.** $\mathrm{SAT}_{\mathsf{pw}}(\log^k |\phi|)$ *is complete for* $\mathsf{NL}[\log^{k-1} n]$*, for* $k \in \mathbb{Z}^+$*, under log-space many-to-one reductions.*

A nondeterministic machine M'' for $\mathrm{SAT}_{\mathsf{pw}}(\log^k |\phi|)$ runs as follows: on its worktape, M'' guesses assignments (each of length $\log^k |\phi|$) for each bag, in the order of path decomposition (storing only the assignments for three bags at any one time). In order to check the correctness of the assignment for the jth bag, the assignments for bags $j-1, j$, and $j+1$ on the working tape, and the consistency of these assignments can be checked in polynomial time. By the properties of path decompositions, checking consistency of consecutive bags is sufficient for correctness. M'' uses $O(\log^k n)$ space and polynomial time. □

Lemma 3.4 and Lemma 3.3 immediately yield the following theorem:

Theorem 3.5. $\mathrm{SAT}_{\mathsf{pw}}(\log^k |\phi|)$ *is complete for* NSC^k*, for* $k \in \mathbb{Z}^+$*, under log-space many-to-one reductions.*

Theorem 3.6. $\mathrm{SAT}_{\mathsf{tw}}(\log^k |\phi|)$ *is complete for* SAC^k_{quasi}*, for* $k \in \mathbb{Z}^+$*, under log-space many-to-one reductions.*

Proof. Containment is by Lemma 3.7 and Lemma 3.2, and hardness is by Lemma 3.8.

Lemma 3.7. $\mathrm{SAT}_{\mathsf{tw}}(\log^k |\phi|) \in \mathsf{NAuxPDA}\big(O(\log^k n), n^{O(1)}\big)$.

Proof. The algorithm witnessing this containment is very natural when expressed as a $\mathsf{NAuxPDA}$; it is a modification of the algorithm in [19] with an additional trick to handle arbitrary CNF clauses, and has a very similar structure to the proof that $\mathrm{SAT}_{\mathsf{pw}}(\log^k |\phi|)$ is in NSC^k.

The $\mathsf{NAuxPDA}$ will perform a depth-first traversal of the tree decomposition, guessing assignments corresponding to the bags (each of length $O(\log^k |\phi|)$) using the worktape and the stack to check consistency of the assignments. More precisely, the $\mathsf{NAuxPDA}$ will start at the root and guess an assignment for the root node, and then recursively search the tree rooted at that node, given the current assignment.

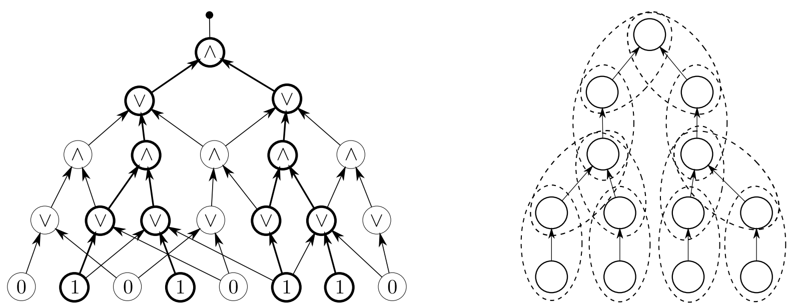
To search the tree rooted at a given node v, given an assignment, the $\mathsf{NAuxPDA}$ will first check if v has any children. If not, the $\mathsf{NAuxPDA}$ will halt and reject if the assignment is not accepting, and otherwise will pop the stack to continue searching the tree rooted at v's parent. Otherwise, the $\mathsf{NAuxPDA}$ will guess assignments for v's children (of which there are ≤ 2), and check that the assignments are consistent, then push the second child and its assignment onto the stack, along with information about v and its assignment, and then search the tree rooted at the first child. When that subtree has been searched, the $\mathsf{NAuxPDA}$ will pop the information for the second child off of the stack and search it. If both subtrees are successfully searched, then the $\mathsf{NAuxPDA}$ pops the stack to continue searching the tree rooted at v's parent.

It can be seen from the description that this machine requires $O(\log^k |\phi|)$ space, and polynomial time. □

Hardness is more interesting. We do a reduction from an arbitrary language in $\mathsf{SAC}^k_{\mathsf{quasi}}$. Similar "generic reductions" (i. e., reducing the computation of families of SAC circuits) for tree-width-related problems have appeared before, e. g., [20] .

Lemma 3.8. $\mathrm{SAT}_{\mathsf{tw}}(\log^k|\phi|)$ *is hard for* SAC^k_{quasi}*, under* LOGSPACE *many-to-one reductions.*

Proof. Fix $L \in \mathsf{SAC}^k_{\mathsf{quasi}}$ and an input x. Let C be the associated SAC circuit, with uniformity realized by a Turing Machine M (i. e., the machine that decides the direct connection language). We construct a formula ϕ that is satisfiable if and only if $C(x) = 1$. Without loss of generality we assume the that the circuit C is of the type constructed in the proof of the first part of Lemma 3.2. In particular, note that we may assume the following normal form for C: (i) C is *layered*, (ii) C is *strictly alternating*: odd-layer gates are OR, even-layer gates are AND, (iii) C has an odd number of layers, and (iv) the AND gates in C have fan-in 2.



(a) A semi-unbounded circuit together with a proof-tree. The NOT gates are assumed to be part of the input

(b) The skeleton of the proof-trees

Figure 1: Proof-tree. In (b), a $\mathrm{SAT}_{\mathsf{tw}}(\log^k|\phi|)$ instance is constructed from the skeleton: each node corresponds to $O(\log^k n)$ Boolean variables; clauses are constructed for each oval with dashed border; and only those variables corresponding to a node shared by different dashed circles must be put into a bag in the tree decomposition. This ensures $O(\log^k n)$ tree-width.

A *proof-tree* is a tree with the same layering as the circuit. Each node of the tree is labelled by an index of a gate from the corresponding layer of the circuit. At odd layers, each node has one child, while at even layers, each node has two children. Two connected nodes must be labelled such that the corresponding gates are connected. At the bottom layer, each node must be labelled by an input gate or a NOT gate which outputs value 1. See Figure 1a for an illustration of an example.

A proof-tree witnesses that $C(x) = 1$. The main observation is that by the above normal form every proof-tree must have the same shape. A *skeleton* is a proof-tree without labels (see Figure 1b). Therefore, $C(x) = 1$ if and only if there exists a labeling to the nodes of the skeleton which turns it into a valid proof-tree. It is important to note that, since C has the form given in the proof of Lemma 3.2, for any node v in the skeleton, all valid labels for v will give v a label corresponding to a gate that is checking the *same* phase with a given stack height; that is, all such labels will correspond to a segment of the computation of a $\mathsf{NAuxPDA}$ with the *same* start and end times. We encode this labeling as a CNF formula as follows. Associate a node v in the skeleton with bit vectors x_v, d_v, t_v, where $|x_v| = |d_v| = \log^k n$, $|t_v|$ is constant. An assignment to these Boolean vectors can be viewed as a labeling in the following sense: x_v indicates the index of the gate, t_v indicates its type, while d_v together with another x_u indicates which predecessor in the circuit it should choose in the proof-tree. More specifically, for every node v at an even-numbered layer in the skeleton with children u_l, u_r we have: $M(\langle n, x_v, 0, \overline{0}, \mathsf{AND}\rangle) = 1$, $M(\langle n, x_v, d_v, x_{u_l}, \mathsf{OR}\rangle) = 1$, and $M(\langle n, x_v, d_v, x_{u_r}, \mathsf{OR}\rangle) = 1$. When v is at an odd-layer, and u is its child, we have $M(\langle n, x_v, 0, \overline{0}, \mathsf{OR}\rangle) = 1$, and either $M(\langle n, x_v, d_v, x_u, \mathsf{AND}\rangle) = 1$ or $M(\langle n, x_v, d_v, x_u, 1\rangle) = 1$.

A correct proof-tree exists if and only if, for each edge (v,u), in the skeleton, the assignments to the variables in x_v, d_v and x_u can be picked so that M accepts the corresponding tuples. This condition can be formalized as $\exists s, M'(s) = 1$, where $|s| = O(\log^k n)$, corresponding to the input bits provided to a Turing machine M' (a modification of M) having running time $O(\log^k n)$ on s. We would like to encode this à la Cook-Levin (see e. g., [3]) as a CNF of size $O(\log^k n)$—but there is a catch. Using the tools provided in [3], this is only possible if M' is *oblivious*—and a naïve approach to making M' oblivious would introduce an unwanted $\log\log n$ factor; thus we need to look more closely at the condition that M' is checking.

M' is taking s as input, and checking that s is giving information about adjacent gates in C. Since all of the valid labels for a node in the skeleton are concerned with the *same* segment of an oblivious $\mathsf{NAuxPDA}$'s computation, we can use the standard technique (e. g., [3]) to build a CNF of size $O(\log^k n)$ verifying that the connectivity information is correct. (For example, s could give the encodings for gates labeled (A,B) and (C,D,E,γ), and we need to verify that the $\mathsf{NAuxPDA}$ can move from A to C pushing γ, and move from E to B popping γ). At the end we take the conjunction of all the CNFs corresponding to the nodes and edges, which is also a CNF F, where F is satisfiable if and only if $C(x) = 1$.

It remains to show that F has tree-width $O(\log^k n)$. Notice that clauses in F are defined for only one specific node, and variables appear in clauses corresponding to at most two nodes. Therefore there is a natural tree decomposition associated with F, as illustrated in Figure 1b, that is, clauses and variables corresponding to an edge in the skeleton form a bag, and two bags are connected when they share variables. By the argument above, this tree decomposition has tree-width $O(\log^k n)$. □□

Remark 3.9. In general, when the tree-width is anything larger than poly-logarithmic, the previous reductions still hold. In particular, $\mathrm{SAT}_{\mathsf{tw}}(w(|\phi|))$ is complete for

$$\mathsf{SAC}\big(O(\log|\phi|), 2^{O(w(|\phi|))}\big).$$

Remark 3.10. The proof of Lemma 3.8 constructs a SAT instance ϕ for which not only the *incidence* graph has small treewidth, but also the *primal graph* has small treewidth. Thus, although the treewidth of the primal graph can be *much* larger than the treewidth of the incidence graph, SAT instances of small treewidth are complete for the $\mathsf{SAC}^k_{\mathsf{quasi}}$ classes, no matter whether treewidth is measured with respect to the incidence graph, as in this paper, or with respect to the primal graph, as in [4].

3.4 Connecting Conjecture 1.1 to complexity theory assumptions and the separation of $\mathrm{SAT}_{\mathsf{pw}}(\log^k|\phi|)$ from $\mathrm{SAT}_{\mathsf{tw}}(\log^k|\phi|)$

We list corollaries of the completeness results obtained in the previous sub-section.

Corollary 3.11. $\mathsf{SAC}^k_{quasi} \not\subseteq \mathsf{TISP}\big(2^{O(\log^k n)}, n^{o(1)}\big) \iff$ *Conjecture 1.1 for tree-width* $O(\log^k|\phi|)$.

In particular, when $k=1$, we have that Conjecture 1.1 for tree-width $O(\log|\phi|)$ is equivalent to

$$\mathsf{SAC}^1 \not\subseteq \mathsf{TISP}\big(n^{O(1)}, n^{o(1)}\big)\,.$$

This corollary is just a resource-scaled form of our initial equivalence for logarithmic tree-width. In fact, by padding[4] we have:

Corollary 3.12. *Conjecture 1.1 for tree-width* $\operatorname{polylog}(|\phi|) \implies \mathsf{SAC}^1 \not\subseteq \mathsf{SC}$.

Thus, modulo these complexity assumptions this settles the lower bound of the Alekhnovich-Razborov question. Note that Corollary 3.12 opens new avenues for propositional proof complexity [6]; i. e., validating our conjecture for restricted types of algorithms implies progress towards $\mathsf{NC} \neq \mathsf{SC}$.

As another corollary, assuming that $\mathsf{NL} \subsetneq \mathsf{SAC}^1$, we separate the complexity of $\mathrm{SAT}_{\mathsf{pw}}$ and $\mathrm{SAT}_{\mathsf{tw}}$.

Corollary 3.13. $\mathrm{SAT}_{\mathsf{tw}}(\log|\phi|)$ *is not log-space reducible to* $\mathrm{SAT}_{\mathsf{pw}}(\log|\phi|)$*, unless* $\mathsf{NL} = \mathsf{SAC}^1$.

In fact, the above holds up to NL-reductions. This corollary extends to every poly-logarithmic width under the scaled assumption $\mathsf{NSC}^k \subsetneq \mathsf{SAC}^k_{\mathsf{quasi}}$. This is the first separation result for width parameterizations of SAT for the same width parameter. Prior to our work there were only results in the opposite direction [19], where some width parameters (e. g., band-width and path-width) were shown to be log-space-equivalent, although combinatorially they can be off by an exponential.

4 Tradeoff algorithms on a single parameter

We consider two basic algorithms. One is time-efficient, which works in time-space

$$\big(2^{2\mathcal{TW}(\phi)}|\phi|^{O(1)}, 2^{\mathcal{TW}(\phi)}|\phi|^{O(1)}\big)\,,$$

whereas the space-efficient one works in time-space

$$\big(3^{\mathcal{TW}(\phi)\log|\phi|}|\phi|^{O(1)}, |\phi|^{O(1)}\big)\,.$$

The first one [35] is the most time-efficient (with respect to the constant in the exponent) algorithm known. The second is our contribution, and it is the first space-efficient algorithm for arbitrary CNFs for tree decompositions on the incidence graph. Our main contribution is combining these two algorithms in a non-trivial way to obtain a tradeoff. Later on, in Section 4.1, we provide a primer to algorithms for SAT instances with given tree decompositions.

[4]Philosophically, the assumption

$$\mathsf{SAC}^k_{\mathsf{quasi}} \not\subseteq \mathsf{TISP}\big(2^{O(\log^k n)}, n^{o(1)}\big)$$

is not really different than the widely-believed assumption $\mathsf{SAC}^1 \not\subseteq \mathsf{TISP}(n^{O(1)}, n^{o(1)})$. By analogy let us consider $\mathsf{P} \neq \mathsf{NP}$ and $\mathsf{E} \neq \mathsf{NE}$. It is true that $\mathsf{E} \neq \mathsf{NE}$ is stronger in the sense that $\mathsf{E} \neq \mathsf{NE}$ implies $\mathsf{P} \neq \mathsf{NP}$ (via a simple padding argument), and it is also the case that at the current state-of-the-art we have no idea how to obtain the converse implication. (In fact, this is true for the vast majority of these resource-scaled analogs of other complexity conjectures). Also, it is worth noting that the converse fails relative to some oracles [11]. However, in principle we see no real reason why one should believe in one and not in the other (especially when the scaling in the resource bounds is moderate); they are merely different manifestations of the same underlying question. Our conjecture is equivalent to $\mathsf{SAC}^1 \not\subseteq \mathsf{TISP}(n^{O(1)}, n^{o(1)})$ for logarithmic tree-width, whereas for larger tree-width we have only shown equivalence to the scaled analogs of $\mathsf{SAC}^1 \not\subseteq \mathsf{TISP}(n^{O(1)}, n^{o(1)})$.

Overview of the time- and space- efficient algorithms. The time-efficient algorithm does dynamic programming using the tree decomposition in a typical way [7]: root the tree to make it a binary tree, then for each bag define a $2^{\mathcal{TW}(\phi)}$ size Boolean array; entry j in the array will be 1 if the subformula rooted at the bag is satisfiable, when the variables are given assignment j, and will be 0 otherwise. Clearly, computing the array for the root will solve the satisfiability of the formula, and indeed by the property of a tree decomposition, the array values can be computed in a leaves-to-root fashion.

To simplify the overview of the space-efficient algorithm we shall temporarily assume that each clause appears in a bag together with all of its variables.[5] Observe that if we fix a truth assignment on a bag, then solving SAT on the given tree decomposition reduces to solving e. g., 3 independent subproblems—think of splitting the degree-3 tree into three subtrees by cutting the original one at this bag. The algorithm works by recursively enumerating and checking truth assignments on the bags. Its performance is determined by the size of the subproblems (ideally all the subtrees have the same size). In Section 4.2 (Lemma 4.2 below) we show that there always exists a good choice for a bag, reminiscent to the well-known "⅓ - ⅔ lemma" for binary trees. The lemmas in Section 4.2 are a bit of an overkill for the analysis of this simple algorithm, but they are also applied in the analysis of the tradeoff.

The tradeoff algorithm: where is the complication? Let us consider for a moment an execution of the space-bounded algorithm. We can visualize each step of the recursion as splitting the tree decomposition at a node (bag)—this bag is replicated at each of the subproblems with the fixed truth assignment. Let the process evolve for a while, and when the forest has enough trees let us single out one such tree. At the boundary (the leaves) of this tree there can be as many as $\log|\phi|$ nodes to which we previously fixed an assignment, i. e., by splitting. *The logarithmically large number of nodes does not affect the performance of the space-efficient algorithm* (at each point of the recursion each bag/node is associated with a single assignment). Now, we switch gears to devise a tradeoff algorithm. A natural thing to do is first to discretize the truth assignment space associated with each bag, say in $2^{(1-\varepsilon)\mathcal{TW}(\phi)}$ many chunks each of size $2^{\varepsilon\mathcal{TW}(\phi)}$, and we perform the recursion as in the space-efficient algorithm but now instead of one assignment we assign the whole chunk. This brings the enumeration, at each recursive step, from $2^{\mathcal{TW}(\phi)}$ down to $2^{(1-\varepsilon)\mathcal{TW}(\phi)}$. On the other hand combining the chunks of the truth assignments into one consistent chunk associated with this tree may increase the space as much as $2^{\varepsilon\log|\phi|\mathcal{TW}(\phi)}$. Overall this is a time-space

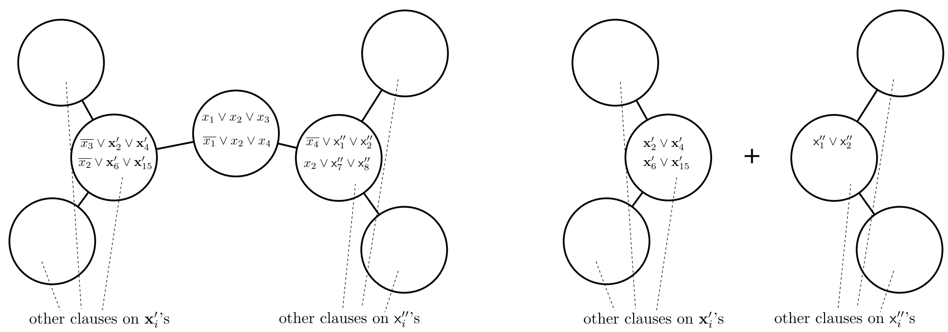
$$\left(2^{(1-\varepsilon)\log|\phi|\mathcal{TW}(\phi)}, 2^{\varepsilon\log|\phi|\mathcal{TW}(\phi)}\right)$$

algorithm, worse both than the time- and space-efficient ones! To devise our tradeoff algorithm we show that it is possible to *simultaneously* (i) perform the splitting in a way that at each step of the execution the forest consists of trees each with at most a constant number of split-nodes and (ii) this splitting results in subproblems of somewhat balanced sizes. Furthermore, we show that it is possible to control the number of splitting nodes per tree in the forest in a way that yields a tradeoff on this parameter (Section 5). This is a different (and competing) tradeoff from the one by the discretization factor ε; i. e., our most general tradeoff algorithm is controlled by two parameters.

[5] We remove this assumption later; see Section 4.1.

4.1 A primer to algorithms for width-parameterized SAT

The structure of a tree decomposition is associated with the concept of separability (see e. g., [8]). Intuitively, the smaller the tree-width is, the easier the graph can be broken into separate components by removing nodes. Separability allows us to devise more efficient algorithms for small tree-width SAT than for general SAT. In some sense, the given tree decomposition allows us to "localize" an exhaustive search. The following example sheds some light on how this can be done towards devising a space-bounded algorithm. Recall that, for this initial overview, we are assuming that all the variables of a clause appear in the same bag with the clauses. We will see later that removing this assumption in a time-efficient manner is non-trivial (in fact, removing it without increasing the base of the exponential running time is an interesting puzzle).



(a) Input tree decomposition.

(b) Fixing an assignment to the variables in the middle bag results in two independent instances.

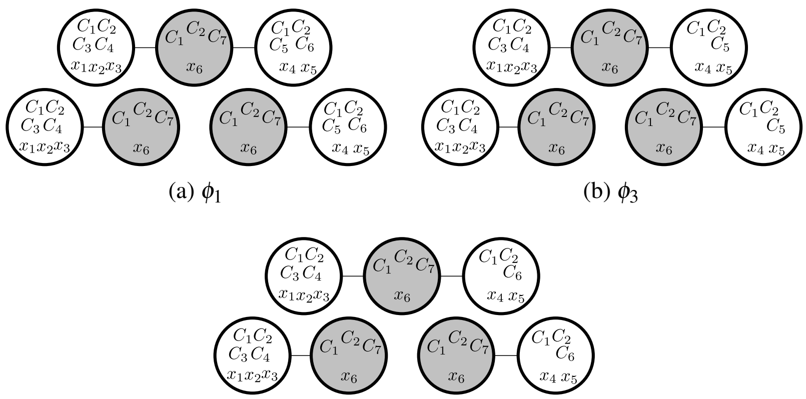
Figure 2: An example showing bounded tree-width SAT can be solved efficiently.

Suppose x_i's, x'_i's and x''_i's are different sets of variables and the tree decomposition is as in Figure 2a. Let us fix a truth assignment to the variables in the bag in the middle, e. g., $x_1 = x_2 = x_3 = x_4 = 1$. Conditioned on this truth assignment we can simplify the instance by removing clauses that are already satisfied, and removing literals in a clause that are set to false. This will result in multiple sub-instances as shown in Figure 2b. The properties of a tree decomposition assure that the sub-instances depend on different sets of variables, i. e., they are *independent*. Since if instead they shared a common variable, this variable would have appeared in the middle bag, e. g., x_2. But this variable is already fixed by the truth assignment.

The satisfiability of the input instance, conditioned on the truth assignment given to the middle bag, is determined by the satisfiability of the two separate sub-instances. Therefore, it suffices to enumerate all truth assignments satisfying all the clauses in the middle bag without causing empty clauses in the simplification phase. Then, recurse into the two independent sub-instances to decide the satisfiability of the original instance. Furthermore, by choosing the middle bag carefully we can invoke this "splitting" on subtrees of somewhat balanced size.

In each recursive step, the most time-consuming part is to enumerate all the assignments satisfying all the clauses in the chosen bag, which costs $O(2^{\mathcal{TW}(\phi)}|\phi|^{O(1)})$ time, and the total running time is $O(2^{\mathcal{TW}(\phi)\log|\phi|}|\phi|^{O(1)})$, which is much better than the current best algorithms for general SAT, which run in time exponential in $|\phi|$.

The subtle additional assumption. The assumption that all variables of a clause appear in the same bag with the clause is not a mild one (especially for CNFs of large cardinality). Of course, in the actual algorithms we make no such assumption. In general, we may have to delay the decision to satisfy a clause. In the above algorithm, we only store the truth assignments to the variables. The following example shows that only storing this information is not enough when aiming at removing the assumption.



(a) ϕ_1 (b) ϕ_3 (c) ϕ_2

Figure 3: Three instances used in the example. Figures on the top are the input tree decompositions, the bottom figures are the two components after fixing assignment to the variables in the middle bag.

Suppose $C_1 = x_1 \vee x_2 \vee x_4 \vee x_6$, $C_2 = \overline{x_1} \vee x_3 \vee x_5$, $C_3 = \overline{x_2}$, $C_4 = \overline{x_3}$, $C_5 = \overline{x_4}$, $C_6 = \overline{x_5}$ and $C_7 = \overline{x_6}$. Three instances ϕ_1, ϕ_2 and ϕ_3 along with their tree decompositions are given in Figure 3, where $\phi_1 = C_1 \wedge \cdots \wedge C_7$, $\phi_2 = C_1 \wedge \cdots \wedge C_5 \wedge C_7$ (i. e., C_6 is missing), and $\phi_3 = C_1 \wedge \cdots \wedge C_4 \wedge C_6 \wedge C_7$ (i. e., C_5 is missing). We say that a clause is satisfied by a literal under a truth assignment if the literal appears in the clause and is set to 1. If an instance is satisfiable, then there is a truth assignment where every clause is satisfied by one of its literals.

Now, consider the splitting operation on the middle bag by fixing a truth assignment to it as above. For all three instances, the only possible assignment for x_6 is 0, since C_7 must be satisfied by $x_6 = 0$. Similarly, in the left bag, we must assign $x_2 = 0$ and $x_3 = 0$ to satisfy C_3 and C_4. In the left bag, the only variable left is x_1, which can satisfy either C_1 or C_2 but not both. The three instances differ in the right part where two variables x_4 and x_5 are left.

Satisfying C_5 requires $x_4 = 0$, which implies that C_1 can not be satisfied by x_4. Similarly, satisfying C_6 requires $x_5 = 0$, so that C_2 can not be satisfied by x_5. In order to find a satisfying truth assignment, when processing the right part, we need information about which of C_1, C_2 is already satisfied in the left part. (Since ϕ_1 is not satisfiable, the final outcome will be the same in either case.) ϕ_2 is satisfied only when C_1

is already satisfied, while ϕ_3 is satisfied only when C_2 is already satisfied. This piece of information is not carried through the middle bag by just the truth assignment to the variables. To overcome this issue we are going to use "clause-bits," which we mentioned briefly in Section 2.3.

4.2 Splitting, consistency, assignment groups

In this section we give some additional notation and technical lemmas which we apply in the analysis of the space-efficient (Section 4.3) and tradeoff algorithms (Sections 4.4 and 5). First we define an operation which allows a natural divide-and-conquer strategy, and a lemma follows the definition for choosing where the operation should occur. Then we define consistency with respect to our definition of assignments, which is somehow subtle and different from consistency of truth assignments. And in the last part of this section, we define a type of *discretized assignment* which is crucial in the tradeoff algorithms.

Definition 4.1 (Splitting operation)**.** Let $T = (V,E)$ be a tree, and $v \in V$. *Splitting* T *at* v is the following operation. Let $T_1,\dots,T_k$ be the trees after removing v from T. The splitting operation results in a forest $\{v\} \cup T_1,\dots,\{v\} \cup T_k$, where $\{v\} \cup T_i$ is the subtree induced by the nodes in T_i together with v. v is called the *splitting node* of this operation.

Given a tree T together with a sequence of splitting operations results in a forest where each subtree in the forest in general has many nodes marked as splitting nodes. Splitting nodes before a specific splitting operation are called *previous splitting nodes*. A splitting operation also splits the set of previous splitting nodes S into S_i's, where S_i is the set of splitting nodes contained in tree T_i, $1 \le i \le k$.

Note that a splitting operation on a tree will result in a forest with more nodes than before, since we duplicate the splitting node and let it appear in each resulting tree. This fact will complicate the analysis of a recursive procedure. To overcome this, consider for each node, we create $d-1$ replicas. When a splitting operation occurs, each replica of the splitting node goes to one of the branches (and redundant ones get removed if there are). Each node can be treated as splitting node only once, so the replicas of a node will be distributed only once. These slightly modified splitting operations will never increase the number of nodes. When analyzing running time on a tree originally with N nodes, one needs to use $d \cdot N$ as a upper bound of the number of nodes. We will see that this is negligible since the number of nodes only appears as a polynomial factor or an argument logarithmically in the exponent of the running time. For ease of exposure, we will stick to the notation N as the number of nodes, while this should be the number after replicating.

A *splitting algorithm* $\mathcal{A}$ computes a function that, given a tree $\mathcal{T}$ together with previous splitting nodes S, returns a node where the next splitting operation is going to be performed. A splitting algorithm formalizes the way of breaking an instance into sub-instances in the space-efficient algorithm. In particular, choosing the *balancing splitting node* is done according to the following lemma.

Lemma 4.2. *Consider a tree of size N, a leaf s and $0 < \alpha < 1$, and satisfying $N > 1/\alpha$. Then, there is a node p where after we split at p, the tree which contains s is of size $\le \lceil \alpha N \rceil$ and every other tree is of size $\le \lceil (1-\alpha)N \rceil$. The node p is called an α-splitting node. Furthermore, such a p can be found in time polynomial in N.*

Proof. We prove this lemma by giving an algorithm for finding p. First root the given tree at s, and then we iteratively construct a path $\langle s \equiv v_1, v_2, \ldots, v_\ell \rangle$ as follows. After constructing the path from v_1 through v_{i-1}, v_i is chosen to be child of v_{i-1} which roots the largest subtree. We claim that there exists an α-splitting node in this path.

Denote by a_i the size of the subtree containing s after splitting at v_i, $1 \le i \le \ell$. It is not hard to see that $a_1 = 1$, $a_\ell = N$, and a_i strictly increases as i increases. Therefore, there must be a j, such that $a_j \le \alpha N$ and $a_{j+1} > \alpha N$. We claim that v_j is the node we need. If $a_{j+1} - a_j = 1$, then splitting at v_j results in two components, where the size of the component containing s is $\lceil \alpha N \rceil$, while the other one is of size $\lceil (1-\alpha)N \rceil$. If $a_{j+1} - a_j > 1$, then there must be a branch at v_j, meaning that v_j has at least two children. Splitting at v_j results in at least three components. One which contains s and is of size smaller than αN. The largest one among the rest is of size smaller than $(1-\alpha)N$. $\square$

Corollary 4.3. *On a bounded-degree tree of size N, there exists a node p, such that after splitting at p each subtree is of size at most $\lceil N/2 \rceil$.*

Consistent assignments. In what follows we assume that there is an initial tree decomposition (recall that the bags are denoted by X_i) together with a sequence of splitting operations S that results in the subtrees along with their splitting nodes.

We refer to an *assignment on a subtree* as the assignment that corresponds only to its splitting nodes. Formally, let $X^* = \bigcup_{v_i \in S} X_i$, and let $\mathcal{V}$ be the variables and $\mathcal{C}$ the clauses which have corresponding nodes in X^*. X^* is the set of variables and clauses on which we define assignments. Suppose in one single splitting operation at the node p, to which X_p is the corresponding bag, $\mathcal{T}$ splits into subtrees $\mathcal{T}_i$'s. Further suppose $R_\mathcal{T}$ is an assignment to $\mathcal{T}$, and $R_{\mathcal{T}_i}$ is an assignment to the subtree $\mathcal{T}_i$. Note that the only difference between $R_\mathcal{T}$ and $R_{\mathcal{T}_i}$'s is at X_p, and $R_\mathcal{T}$ and $R_{\mathcal{T}_i}$'s are said to be *consistent* if

(1) for a variable x:

  a) if x appears in X_p, then all the bits for x in each $R_{\mathcal{T}_i}$ are assigned to the same value;

  b) otherwise, all the bits for x are assigned to the same value as in $R_\mathcal{T}$;

(2) for a clause C:

  a) if C appears in X^* and is assigned to 0, then $\forall i$ every bit for C in $R_{\mathcal{T}_i}$ is assigned to 0;

  b) otherwise, $\exists$ exactly one i such that in $R_{\mathcal{T}_i}$ the bit corresponding to C is assigned to 1.

Remark 4.4. The latter point in the definition, where in exactly one of the subtrees we require that the corresponding bit equals to 1, is somewhat subtle. The following lemma crucially depends on this property.

Lemma 4.5. *For every assignment $\mathcal{R}_\mathcal{T}$ to the tree $\mathcal{T}$, the number of assignments $R_{\mathcal{T}_i}$ to subtrees $\mathcal{T}_i$'s consistent with $\mathcal{R}_\mathcal{T}$ is at most $d^{\mathcal{TW}(\phi)}$. (Recall $d \in \{2,3\}$.)*

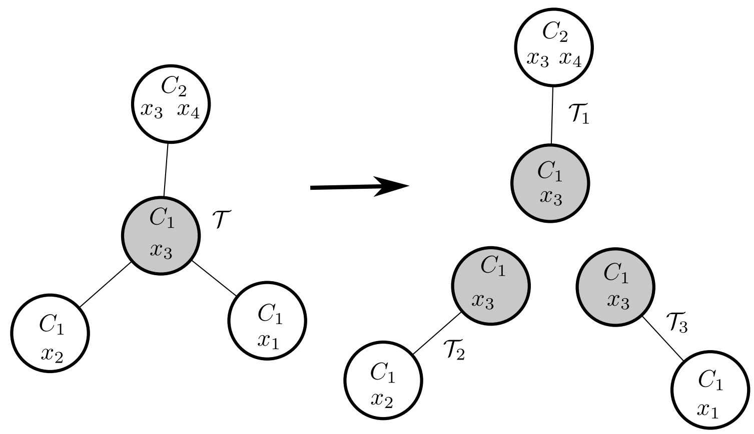


Figure 4: Consistent assignments. $C_1 = x_1 \vee \overline{x_2} \vee x_3$, $C_2 = x_3 \vee \overline{x_4}$. Consider splitting at the gray bag, while fixing the value of the bits, 1 for C_1, 0 for x_3. Any possible consistent assignments for $\mathcal{T}_1$, $\mathcal{T}_2$, $\mathcal{T}_3$ will have 0 for x_3; in this consistent assignment C_1 has value 1 in $\mathcal{T}_1$, and has value 0 in $\mathcal{T}_2$ and $\mathcal{T}_3$.

Proof. Let X_p be the bag corresponding to the splitting node p. For each variable x in the bag X_p, there are 2 possible assignments of the bit for x in the $\mathcal{T}_i$'s. For each clause C in X_p, if C appears in $R_{\mathcal{T}}$ and is assigned to 0, by the definition of consistency, all the bits for C in the $\mathcal{T}_i$'s are assigned to 0. Otherwise, in exactly one $\mathcal{T}_i$, the bit for C is assigned to 1; in this case there are at most d valid assignments. Recall that d is the maximum degree of the tree decomposition. □

We define a satisfying assignment in a way consistent with the role of clause bits in the assignments.

Definition 4.6. For a tree $\mathcal{T}$ with splitting nodes S, an assignment $R_{\mathcal{T}}$ is *satisfying* if there exists a truth assignment A to every variable in $\mathcal{T}$, such that

(1) every truth value for a variable in $R_{\mathcal{T}}$ agrees with the corresponding value in A,

(2) every clause C that appears in $\mathcal{T}$ where C does not appear in S, is satisfied by A,

(3) every clause C that appears in S and the corresponding bit is assigned to 1 by $R_{\mathcal{T}}$ is satisfied by A.

A satisfying assignment of the input tree decomposition with empty splitting nodeset implies that the input formula is satisfiable. The following lemma shows that the task of finding a satisfying assignment can be done recursively.

Lemma 4.7. *An assignment $R_{\mathcal{T}}$ is* satisfying *if and only if there exist a* satisfying *assignment $R_{\mathcal{T}_i}$ to each subtree $\mathcal{T}_i$, such that all of the assignments $R_{\mathcal{T}_i}$ are consistent with $R_{\mathcal{T}}$.*

Proof. For a tree $\mathcal{T}$ with splitting nodes S, suppose that splitting at node p results in the subtrees $\{\mathcal{T}_i\}$. Suppose that the assignment $R_{\mathcal{T}}$ is *satisfying*. By Definition 4.6, there exists a truth assignment on variables within $\mathcal{T}$ that makes all of the clauses true. This truth assignment induces assignments $R_{\mathcal{T}_i}$

consistent with $R_{\mathcal{T}}$, such that for these truth assignments the conditions in Definition 4.6 are met. (Some of the clause bits in the $R_{\mathcal{T}_i}$ may need to be set to zero, to ensure consistency.)

For the other direction suppose that there exist assignments $R_{\mathcal{T}_i}$ of the subtrees $\mathcal{T}_i$, such that the assignments $R_{\mathcal{T}_i}$'s are consistent with $R_{\mathcal{T}}$ and all $R_{\mathcal{T}_i}$'s are *satisfying*. For each subtree $\mathcal{T}_i$, there exists a truth assignment complying to Definition 4.6. Since all these truth assignments agree on their common variables (because the common variables appear in splitting nodes in S), we can get a truth assignment from their union, which also meets the axioms in Definition 4.6. Therefore, the assignment $R_{\mathcal{T}}$ is satisfying. □

An ε*-assignment group* ε-$\mathsf{GR}_{\mathcal{T}}$ is a set of all possible assignments to S, where at most $(1-\varepsilon)|S|\mathcal{TW}(\phi)$ entries are fixed. By definition, ε-assignment groups can be identified by the fixed entries. Consider a tree $\mathcal{T}$, ε-assignment group ε-$\mathsf{GR}_{\mathcal{T}}$, and subtrees $\mathcal{T}_i$ resulting from a split at some node p. Consider one such subtree $\mathcal{T}_i$; let S_i be the set of splitting nodes for $\mathcal{T}_i$, and note that $S_i \subseteq S \cup \{p\}$ (and very likely it is a proper subset). By fixing the "first" $(1-\varepsilon)\mathcal{TW}(\phi)$ entries corresponding to variables or clauses contained in the node $p \in S_i$ (using some fixed ordering), one obtains an ε-assignment group ε-$\mathsf{GR}_{\mathcal{T}_i}$ for $\mathcal{T}_i$.

Given a tree $\mathcal{T}$ and subtrees $\mathcal{T}_i$ resulting from a split, the ε-assignment groups ε-$\mathsf{GR}_{\mathcal{T}}$ and ε-$\mathsf{GR}_{\mathcal{T}_i}$'s are called *consistent* if there exist $R_{\mathcal{T}} \in \varepsilon$-$\mathsf{GR}_{\mathcal{T}}$ and $R_{\mathcal{T}_i} \in \varepsilon$-$\mathsf{GR}_{\mathcal{T}_i}$ for each i, such that $R_{\mathcal{T}}$ and the $R_{\mathcal{T}_i}$'s are consistent.

Note that the fixed entries for the splitting node p may be different among subtrees (because of the rules regarding clause bits), and note also that some of the unfixed entries in $\mathcal{T}$ may fixed in subtrees (because they appear in p). The following important lemma holds, which basically generalizes Lemma 4.5.

$$\underbrace{1\ 0\ 1\ 0\ 1\ 1\ 1}_{(1-\varepsilon)|\mathcal{S}|\mathcal{TW}(\phi)}\quad \underbrace{*\ *\ *}_{\varepsilon|\mathcal{S}|\mathcal{TW}(\phi)}$$

(a) An ε-assignment group where $(1-\varepsilon)$ fraction of values are fixed

$$\begin{array}{c} 1\ 0\ 1\ 0\ 1\ 1\ 1\ 0\ 0\ 0 \\ 1\ 0\ 1\ 0\ 1\ 1\ 1\ 0\ 0\ 1 \\ \vdots \\ 1\ 0\ 1\ 0\ 1\ 1\ 1\ 1\ 1\ 1 \end{array}$$

(b) Assignments in the group

Figure 5: ε-assignment group.

Lemma 4.8. *The number of distinct* ε-$GR_{\mathcal{T}_i}$*'s consistent with* ε-$GR_{\mathcal{T}}$ *is at most* $d^{(1-\varepsilon)\mathcal{TW}(\phi)}$.

Proof. Here we only consider the first $(1-\varepsilon)$ fraction of entries, which are going to be fixed, since as pointed out in previous discussion, an ε-assignment group is identified by the values of the fixed entries. For each variable x, there are $2(\leq d)$ possible values. For each clause C, let $d_0(\leq d)$ be the number of subtrees created by splitting at p. There are two different cases:

(1) *C is not in any previous splitting nodes, or C is in some previous splitting node but its value is unfixed.* There are d_0 possible ways of assigning values to the bit for C, such that there is exactly one of $\mathcal{T}_i$'s, whose bit for C is set to 1 to ensure C is satisfied.

(2) *C is in some previous splitting node, and its value is fixed in ε-$GR_{\mathcal{T}}$.* If the bit for C is assigned 1, then there are d_0 possible assignments to C similar as above, otherwise the only possible way is to set all bits for C to 0.

Since there are at most $(1-\varepsilon)\mathcal{TW}(\phi)$ entries that need to be fixed, in order to form the ε-$\mathsf{GR}_{\mathcal{T}_i}$'s, the number of different combinations of ε-$\mathsf{GR}_{\mathcal{T}_i}$'s consistent with ε-$\mathsf{GR}_{\mathcal{T}}$ is at most $d^{(1-\varepsilon)\mathcal{TW}(\phi)}$. □

4.3 The space-efficient algorithm

The space-efficient algorithm is described in Algorithm 2. $\mathcal{T}$ is a tree with previous splitting nodes S, and $R_{\mathcal{T}}$ is the assignment fixed on the tree. A subtle point that affects the running time of this algorithm is addressed in Remark 4.4. The correctness of the algorithm directly follows by Lemma 4.7.

Algorithm 2 SAT($\mathcal{T}$, $R_{\mathcal{T}}$).

```
if all nodes in 𝒯 are previous splitting nodes then
    if every clause in R_𝒯 assigned value 1 is satisfied by some variable in 𝒯 then
        return True
    else
        return False
    end if
else
    split at the 1/2-splitting node, and denote the subtrees as 𝒯_i's
    for all R_{𝒯_i}'s consistent with R_𝒯 do
        if ∀𝒯_i, SAT(𝒯_i, R_{𝒯_i}) = True then
            return True
        end if
    end for
    return False
end if
```

This algorithm requires only $|\phi|^{O(1)}$ space, because there are only $O(\log|\phi|)$ assignments to be stored during the process. Suppose $T(N)$ is the running time on a decomposition with N nodes. By Lemma 4.5

$$T(N) \le O\left(d^{\mathcal{TW}(\phi)}\right) T\left(\frac{1}{2}N\right) + |\phi|^{O(1)}$$

that is, $T(|\phi|) = O\left(d^{\mathcal{TW}(\phi)\log|\phi|}|\phi|^{O(1)}\right)$, where by the normal form assumption $d = 3$, i. e.,

$$T(|\phi|) = 3^{\mathcal{TW}(\phi)\log|\phi|}|\phi|^{O(1)}.$$

4.4 Tradeoff algorithms

We present a family of algorithms for bounded tree-width SAT described in Algorithm 3. Different implementations of SPLITTING ALG (line 10) result in different algorithms. SAT-TRADEOFF is a procedure that takes a tree decomposition $\mathcal{T}$, a previous splitting node set $\mathcal{S}$, and an ε-assignment group $\varepsilon\text{-GR}_{\mathcal{T}}$, and returns an array $M(\mathcal{T},\varepsilon\text{-GR}_{\mathcal{T}})$ of $2^{\varepsilon|\mathcal{S}|\mathcal{TW}(\phi)}$ entries, where the ith entry indicates whether the i-th assignment of $\varepsilon\text{-GR}_{\mathcal{T}}$ can be extended to a satisfying truth assignment.

Algorithm 3 SAT-TRADEOFF($\mathcal{T}$, $\mathcal{S}$, $\varepsilon\text{-GR}_{\mathcal{T}}$)

$M(\mathcal{T},\varepsilon\text{-GR}_{\mathcal{T}}) \leftarrow$ all-zero array
if all nodes in $\mathcal{T}$ are in $\mathcal{S}$ **then**
    for all $j : 1 \le j \le |\varepsilon\text{-GR}_{\mathcal{T}}|$ **do**
        $R_{\mathcal{T}} \leftarrow j$th assignment in $\varepsilon\text{-GR}_{\mathcal{T}}$
        if every clause whose bit in $R_{\mathcal{T}}$ assigned 1 is satisfied by some variable in $\mathcal{T}$ **then**
            $M(\mathcal{T},\varepsilon\text{-GR}_{\mathcal{T}})_j \leftarrow 1$
        end if
    end for
else
    split at SPLITTING ALG($\mathcal{T}$, $\mathcal{S}$), and denote the subtrees $\mathcal{T}_i$'s ▷ Replaceable
    for all $\varepsilon\text{-GR}_{\mathcal{T}_i}$'s consistent with $\varepsilon\text{-GR}_{\mathcal{T}}$ by fixing $(1-\varepsilon)\mathcal{TW}(\phi)$ entries **do**
        $\forall i$, $M(\mathcal{T}_i,\varepsilon\text{-GR}_{\mathcal{T}_i}) \leftarrow$ SAT-tradeoff$(\mathcal{T},\mathcal{T}_i,\varepsilon\text{-GR}_{\mathcal{T}_i})$
        for all $j : 1 \le j \le |\varepsilon\text{-GR}_{\mathcal{T}}|$ **do**
            $R_{\mathcal{T}} \leftarrow j$th assignment in $\varepsilon\text{-GR}_{\mathcal{T}}$
            for all $R_{\mathcal{T}_i}$'s chosen $\varepsilon\text{-GR}_{\mathcal{T}_i}$'s correspondingly **do**
                if $\forall i$, $M(\mathcal{T}_i,R_{\mathcal{T}_i}) = 1$ **and** $\forall \mathcal{T}_i$, $R_{\mathcal{T}_i}$'s are consistent with $R_{\mathcal{T}}$ **then**
                    $M(\mathcal{T},\varepsilon\text{-GR}_{\mathcal{T}})_j \leftarrow 1$
                end if
            end for
        end for
    end for
end if
return $M(\mathcal{T},\varepsilon\text{-GR}_{\mathcal{T}})$

A $type_\ell$ tree is a tree with ℓ previous splitting nodes. Let α be a parameter satisfying $0 < \alpha < 1/2$. The splitting algorithm $\mathcal{H}_2$ described in Algorithm 4 has the property that it never creates a $type_\ell$ tree, for any $\ell \ge 3$.

The performance of the tradeoff algorithms is not hard to analyze tightly (unlike the rather involved analysis of the two-parameter generalized tradeoff in Section 5), and it is summarized in the following theorem.

Theorem 4.9. SAT *of tree-width* $\mathcal{TW}(\phi)$ *can be solved in simultaneously*

$$O\big(d^{1.441(1-\varepsilon)\mathcal{TW}(\phi)\log|\phi|}|\phi|^{O(1)}\big)$$

time and

$$O\big(2^{2\varepsilon\mathcal{TW}(\phi)}|\phi|^{O(1)}\big)$$

space, where ε *is a free parameter,* $0 < \varepsilon < 1$.

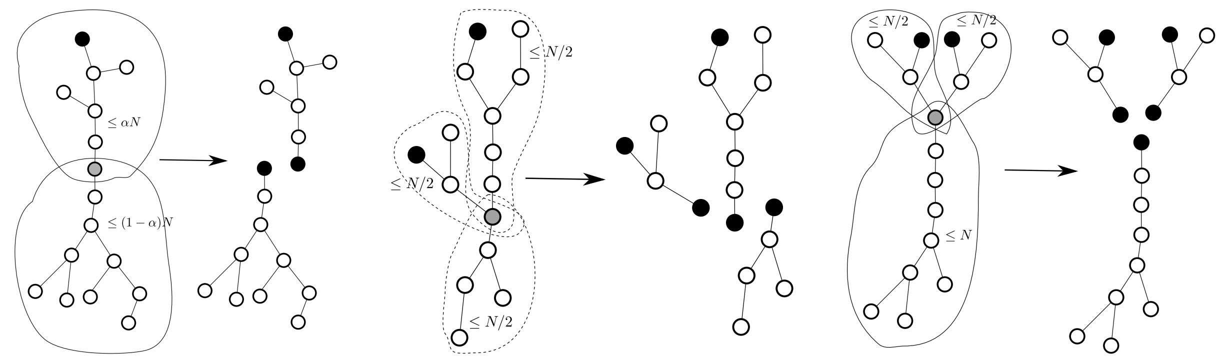


Figure 6: Choosing splitting node in three different cases. Rounding errors are ignored for simplicity.

Proof. Denote by $T_1(N)$, $T_2(N)$ the running time of Algorithm 4 using splitting rule $\mathcal{H}_2$ on $type_1$ or $type_2$ trees each of N nodes respectively. Splitting a $type_1$ tree results in multiple $type_1$ trees with size at most $(1-\alpha)N$ and one $type_2$ tree with size at most αN, so we have

$$T_1(N) \leq O(d^{(1-\varepsilon)\mathcal{TW}(\phi)})\left(T_1\left((1-\alpha)N\right)+T_2\left(\alpha N\right)\right)+2^{O(\mathcal{TW}(\phi))}.$$

Splitting a $type_2$ tree, when the $1/2$-splitting is on the path between p_1 and p_2 results in two $type_2$ trees with size at most $N/2$ and multiple $type_1$ trees. Otherwise, the splitting operation results in two $type_2$ trees with size at most $N/2$ and several $type_1$ trees. Hence:

$$T_2(N) \leq O(d^{(1-\varepsilon)\mathcal{TW}(\phi)})\left(T_1(N)+T_2(N/2)\right)+2^{O(\mathcal{TW}(\phi))}.$$

Set $\alpha = \frac{3-\sqrt{5}}{2}$ to minimize the values of $T_1(N)$ and $T_2(N)$, we have

$$\begin{aligned} T_1(N) &\leq O(d^{(1-\varepsilon)\mathcal{TW}(\phi)})\left(T_1\left((1-\alpha)N\right)+T_2\left(\alpha N\right)\right)+2^{O(\mathcal{TW}(\phi))} \\ &\leq O(d^{(1-\varepsilon)\mathcal{TW}(\phi)})T_1\left((1-\alpha)N\right)+O(d^{2(1-\varepsilon)\mathcal{TW}(\phi)})T_1\left(\alpha N\right)+2^{O(\mathcal{TW}(\phi))}. \end{aligned}$$

Therefore:

$$T_1(N) \leq d^{\frac{1}{-\log(1-\alpha)}(1-\varepsilon)\mathcal{TW}(\phi)\log N}|\phi|^{O(1)}.$$

Since $type_i$, $i \geq 3$ trees are not allowed, the space requirement is $2^{2\varepsilon\mathcal{TW}(\phi)}|\phi|^{O(1)}$. □

4.5 Optimality of the splitting algorithm for the single-parameter tradeoff

The splitting algorithm presented above is a specific one, with the property that it does not create $type_i$ trees for any $i \geq 3$. Interestingly, it can be shown that this specific splitting algorithm is optimal over all splitting strategies which enjoy this property.

Definition 4.10. Denote by $\mathfrak{A}_c$ ($\forall c \geq 2$) the family of algorithms for SAT with bounded tree-width following the framework in Algorithm 3 which use a splitting algorithm without creating $type_i$ trees $\forall i > c$.

Algorithm 4 $\mathcal{H}_2(\mathcal{T},\mathcal{S})$

```
if T with S is a type_0 tree then
    return the 1/2-splitting node
else if T with S is a type_1 tree then
    regard the previous splitting node as root
    return the α-splitting node                          ▷ Figure 6a
else if T with S is a type_2 tree then
    suppose S = {p_1, p_2}
    regard p_1 as root and compute the 1/2-splitting node q
    if q is on the path between p_1 and p_2 then
        return q                                         ▷ Figure 6b
    else
        return the least common ancestor of q and p_2    ▷ Figure 6c
    end if
end if
```

We lower bound the running time of all algorithms in $\mathfrak{A}_2$ by showing hard instances based on generalizations of Fibonacci trees.

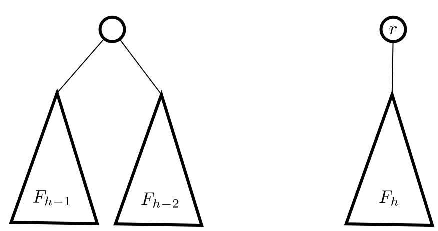
Definition 4.11. For any positive integer h, a *h-Fibonacci tree*(denoted as F_h) is a rooted tree recursively defined as following,

(1) if $h = 1$, F_h contains only 1 node;

(2) if $h = 2$, F_h contains 2 nodes and one edge between them;

(3) if $h > 2$, F_h is constructed by a root connecting roots of two subtrees F_{h-2} and F_{h-1}.

An *extended* (h,r)*-Fibonacci tree* (denote as $F^*_{h,r}$) is constructed by adding one edge between a root node r and the root of subtree F_h.



(a) F_h (b) $F^*_{h,r}$

Figure 7: Fibonacci tree and extended Fibonacci tree.

In what follows, we focus on the structure of the trees and inspect the running time of an algorithm in $\mathfrak{A}_2$ on a formula with a tree decomposition with the specific structure, and omit the details of constructing a formula having tree decomposition of a certain structure here. Consider an extended $(h+2,r)$-Fibonacci tree $F^*_{h+2,r}$ with N nodes, $h = \lceil \log_{(1+\sqrt{5})/2} N \rceil - 2$. We prove that this is hard for any algorithms in $\mathfrak{A}_2$, namely,

Theorem 4.12. *Every algorithm in $\mathfrak{A}_2$ runs in $\Omega\big(3^{1.441(1-\varepsilon)\mathcal{TW}(\phi)\log N}|\phi|^{\Theta(1)}\big)$ time on $F^*_{h+2,r}$.*

Before giving the proof of the theorem, we define two types of trees which appear in intermediate phases of the splitting algorithm and lower bound the running time on these trees: $\mathcal{T}_{1,h}$ (a special *type*$_1$ tree) is constructed by a splitting node connected to the root of a subtree F_h (same shape as $F^*_{h,r}$, the node at the position of r is a splitting node); and $\mathcal{T}_{2,h}$ (a special *type*$_2$ tree) is constructed by two separate splitting nodes connected to a node r which roots a subtree $F^*_{h,r}$.

Lemma 4.13. *Every algorithm in $\mathfrak{A}_2$ runs in $\Omega\big(3^{(1-\varepsilon)\mathcal{TW}(\phi)h}|\phi|^{\Theta(1)}\big)$ time on $\mathcal{T}_{1,h}$, and runs on $\mathcal{T}_{2,h}$ in time $\Omega\big(3^{(1-\varepsilon)\mathcal{TW}(\phi)(h+1)}|\phi|^{\Theta(1)}\big)$.*

Proof. The proof is by induction on h. Base cases are vacuous where $h \le 2$. Suppose the statement is true for any $h_0 < h$, consider the induction steps:

(1) For $\mathcal{T}_{1,h}$, if we split at the root of F_h, a $\mathcal{T}_{1,h-1}$ will be derived, the running time on which is

$$\Omega\big(3^{(1-\varepsilon)\mathcal{TW}(\phi)(h-1)}|\phi|^{\Theta(1)}\big)\,;$$

otherwise if we split at some node inside the subtrees F_{h-1} or F_{h-2}, the running time on the subtree containing two splitting nodes (a new one and a previous one) is lower bounded by the running time of a $\mathcal{T}_{2,h-2}$, which is

$$\Omega\big(3^{(1-\varepsilon)\mathcal{TW}(\phi)((h-2)+1)}|\phi|^{\Theta(1)}\big)\,.$$

To see this, assume for example that the splitting node p is inside the F_{h-1}. Imagine that all the nodes in F_{h-1} except those on the path between p and the node v which connects to the root of F_{h-2} are pruned for free, and then replace the path between p and v by an edge between them. A $\mathcal{T}_{2,h-2}$ is derived this way, and note that the running time will not increase after the procedure. Observe that enumerating all assignments for a newly created splitting node requires time

$$\Omega\big(3^{(1-\varepsilon)\mathcal{TW}(\phi)}|\phi|^{\Theta(1)}\big)\,,$$

so the running time for $\mathcal{T}_{1,h}$ is

$$\Omega\big(3^{(1-\varepsilon)\mathcal{TW}(\phi)h}|\phi|^{\Theta(1)}\big)\,.$$

(2) For $\mathcal{T}_{2,h}$, the splitting must occur at the node connecting the two splitting nodes, which creates a $\mathcal{T}_{1,h}$, the running time on which as shown above is

$$\Omega\big(3^{(1-\varepsilon)\mathcal{TW}(\phi)h}|\phi|^{\Theta(1)}\big)\,.$$

Taking the time required for enumerating all assignments for the new splitting node into consideration, the total running time for $\mathcal{T}_{2,h}$ is therefore lower bounded by

$$\Omega\big(3^{(1-\varepsilon)\mathcal{TW}(\phi)(h+1)}|\phi|^{\Theta(1)}\big)\,. \qquad \square$$

Proof of Theorem 4.12. The very first step of splitting at any node of $F^*_{h+2,r}$ will result in a $type_1$ tree, the running time on which is lower bounded by the running time on $\mathcal{T}_{1,h}$. By Lemma 4.13, that is

$$\Omega\big(3^{\frac{1}{-\log(1-\alpha)}(1-\varepsilon)\mathcal{TW}(\phi)\log N}|\phi|^{\Theta(1)}\big),$$

where

$$N = \Theta\left(\left(\frac{1+\sqrt{5}}{2}\right)^h\right), \quad \text{and} \quad \alpha = \frac{3-\sqrt{5}}{2}$$

(which matches the parameter chosen in the tradeoff algorithm). By simplifying this expression we obtain the theorem. □

5 Generalized two-parameter tradeoff algorithms

In this section we establish Theorem 5.1 below, by exhibiting a family of algorithms that achieve time-space tradeoffs generalizing the algorithms in the previous section. Each algorithm in this family is identified by the parameters (ε, c). Moreover, we show that both of these parameters are necessary to achieve different time-space tradeoffs. Intuitively, parameter $0 < \varepsilon < 1$ corresponds to the granularity of the discretization of the assignment space, whereas the integer parameter $c \geq 2$ has to do with the "complexity" of the rule applied recursively during the truth assignment search.

Theorem 5.1. *For every integer $c \geq 2$ and ε, where $0 < \varepsilon < 1$, a* SAT *instance ϕ with a tree decomposition of width $\mathcal{TW}(\phi)$ and N nodes, can be decided in time-space*

$$\Big(3^{(\lambda_c(\log N - c)+c)(1-\varepsilon)\mathcal{TW}(\phi)}|\phi|^{O(1)},\ 2^{c\varepsilon\mathcal{TW}(\phi)}|\phi|^{O(1)}\Big)$$

for a constant λ_c.

λ_c is a constant depending on c. To be more specific, λ_c is defined as $-\log x_c$, where x_c is the root with largest absolute value of the polynomial equation: $X^c - X^{c-1} - X^{c-2} - \cdots - 1 = 0$. The first few values of λ_c for small c's are listed in Table 1.

c	2	3	4	5	6
λ_c	1.441	1.138	1.057	1.026	1.013

Table 1: Values of λ_c for small c's.

5.1 Generalized tradeoff algorithms

We have already seen a tradeoff algorithm which avoids $type_i$ trees for $i \geq 3$. It is natural to ask if the algorithm can be generalized to allow up to $type_c$ trees for any fixed $c \geq 2$, and more importantly if by doing so there is any gain in the running time (clearly, there will be a loss in the space). Indeed, this is possible and as c increases the running time decreases while the space requirement increases.

First, we generalize the splitting algorithm to allow $type_i$ trees for i up to c. For arbitrary $1 \le i \le c$, consider splitting a $type_i$ tree: suppose the splitting node is p. If p is on the path between some pair of previous splitting nodes, splitting at this node results in several $type_j$ $(j \le i)$ trees; otherwise, splitting results in several $type_1$ trees and one $type_{i+1}$ tree. Formally, we devise an algorithm $\mathcal{H}_c$, such that when splitting a $type_i$ tree, we invoke $\mathcal{H}_c$ to determine the splitting node. This is an implementation of SPLITTING ALG in Algorithm 3.

Algorithm 5 $\mathcal{H}_c(\mathcal{T}, \mathcal{S})$

```
if $\mathcal{T}$ with $\mathcal{S}$ is a $type_0$ tree then
    return the $1/2$-splitting node
else
    suppose $\mathcal{T}$ with $\mathcal{S}$ is a $type_i$ tree
    if $|\mathcal{T}| \le 2^{c-i}$ then
        return the $1/2$-splitting node
    else
        pick an arbitrary node from $\mathcal{S}$ as root and compute the $\alpha_{c,i}$-splitting node $q_1$
        if $q_1$ is not on the path between any pair in $\mathcal{S}$ then
            return $q_1$                                                        ▷ Figure 8a
        else
            compute a $1/2$-splitting node $q_2$.
            if $q_2$ is on the path between any pair in $\mathcal{S}$ then
                return $q_2$                                                    ▷ Figure 8b
            else
                return the least common ancestor of $q_2$ and all nodes in $\mathcal{S}$   ▷ Figure 8c
            end if
        end if
    end if
end if
```

Each $\alpha_{c,i}$ for any $1 \le i < c$ is a parameter satisfying $0 \le \alpha_{c,i} \le 1/2$. To prevent $type_{c+1}$ trees, splitting node of a $type_c$ tree must be on the path between some pair of previous splitting nodes, this is assured by setting $\alpha_{c,c} = 0$. For a fixed c, the running time and space of the algorithm solving SAT of bounded tree-width utilizing the splitting algorithm $\mathcal{A}$ are summarized in Theorem 5.1 (see page 326).

We introduce the following notation in order to discuss the splitting depth.

Definition 5.2. The *c-splitting depth* $\mathsf{SD}_c(\mathcal{A}, \mathcal{T}, S)$ of a *splitting algorithm* $\mathcal{A}$ on tree $\mathcal{T}$ with previous splitting nodes S is inductively defined as follows (where the case for $|S| > c$ is arbitrary):

$$\mathsf{SD}_c(\mathcal{A}, \mathcal{T}, S) = \begin{cases} \max_{(\mathcal{T}_0, S_0) \in C_{\mathcal{T},S,p}} \mathsf{SD}_c(\mathcal{A}, \mathcal{T}_0, S_0) + 1 & |S| \le c, |S| < |\mathcal{T}|, \\ 0 & |S| \le c, |S| = |\mathcal{T}|, \\ \infty & |S| > c, \end{cases}$$

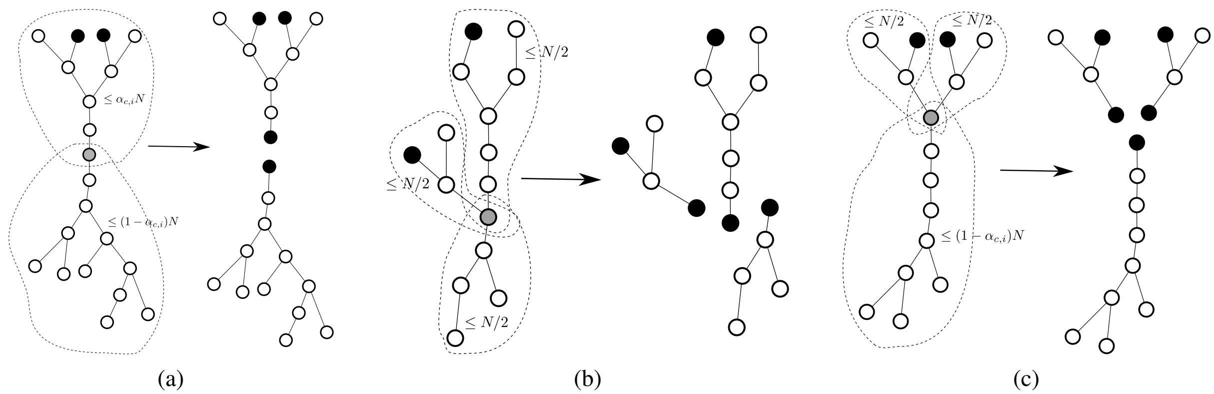


Figure 8: Choosing splitting node for a $type_i$ tree ($i = 2$ here) in three different cases. Rounding errors are ignored for simplicity.

where p is the output of $\mathcal{A}$ on $\mathcal{T}$ and S, $C_{\mathcal{T},S,p}$ is the set of subtrees by splitting at p in tree $\mathcal{T}$ with previous splitting nodes S.

We define the *c-minimum splitting depth* $\mathsf{minSD}_c(\mathcal{T},S)$ to be the minimum value of $\mathsf{SD}_c(\mathcal{A},\mathcal{T},S)$, over all splitting algorithms.

Under this notation, given a tree $\mathcal{T}$, any algorithm $\mathcal{A}$ avoiding $type_{c+1}$ trees requires time

$$d^{(1-\varepsilon)\mathsf{SD}_c(\mathcal{A},\mathcal{T},\emptyset)\mathcal{TW}(\phi)}|\phi|^{O(1)}$$

and space

$$2^{c\varepsilon\mathcal{TW}(\phi)}|\phi|^{O(1)}.$$

In fact, bounding the running time is a non-trivial issue (the derived recurrences are in a "perplexed" form). The proof of Theorem 5.1 follows by two technical lemmas: Lemma 5.3 establishes the recurrences according to the recursive algorithm, and Lemma 5.4 deals with choice of parameters. For simplicity of presentation we ignore issues regarding the divisibility of N by 2.

Lemma 5.3. *For every $c \geq 2$, any tree $\mathcal{T}$ with N nodes and splitting node set S of size i, let*

$$D_{c,i}(N) = \max_{\mathcal{T},S} \mathsf{SD}_c(\mathcal{H}_c,\mathcal{T},S).$$

Then,

$$\begin{cases} D_{c,i}(N) \leq \max\{D_{c,1}\left((1-\alpha_{c,i})N\right), D_{c,i+1}\left(\alpha_{c,i}N\right), D_{c,i}\left(N/2\right)\} + 1 & \forall i : 1 \leq i < c\,, \\ D_{c,c}(N) \leq \max\{D_{c,1}(N), D_{c,c}(N/2)\} + 1\,. \end{cases}$$

Proof. Without loss of generality, suppose $N \geq 2^c$. Consider splitting a $type_i$ tree with splitting nodes S, $1 \leq i < c$. If the $\alpha_{c,i}$-splitting-node m is not on the path between any pair of previous splitting nodes, splitting at m will result in multiple $type_1$ trees of size at most $\lceil (1-\alpha_{c,i})N \rceil$ and one $type_{i+1}$ tree of size at most $\lceil \alpha_{c,i}N \rceil$. Otherwise, since $1-\alpha_{c,i} \geq 1/2$, the maximal possible size of a $type_1$ tree created by any splitting node will not exceed $\lceil (1-\alpha_{c,i})N \rceil$. Splitting at the 1/2-splitting-node c will result in multiple $type_j (j \leq i)$ trees of size at most $\lceil N/2 \rceil$, otherwise, splitting at the least common ancestor of c and all previous splitting nodes as p, will result in multiple $type_1$ tree of size at most $\lceil (1-\alpha_{c,i})N \rceil$ and many $type_j (j \leq i)$ trees with size at most $\lceil N/2 \rceil$. In summary,

$$D_{c,i}(N) \leq \max\{D_{c,1}\left((1-\alpha_{c,i})N\right), D_{c,i+1}\left(\alpha_{c,i}N\right), D_{c,i}\left(N/2\right)\} + 1\,.$$

Now, consider splitting a $type_c$ tree with splitting nodes S. Since $\alpha_{c,c} = 0$, we always ignore the $(1-\alpha_{c,i})$-splitting-node m. Splitting at the 1/2-splitting-node c will result in multiple $type_j (j \leq i)$ trees of size at most $\lceil N/2 \rceil$. Splitting at the least common ancestor of c and all previous splitting nodes will result in multiple $type_1$ tree with size at most N and multiple $type_j (j \leq i)$ trees with size at most $\lceil N/2 \rceil$, namely:

$$D_{c,c}(N) \leq \max\{D_{c,1}(N), D_{c,c}(N/2)\} + 1\,. \qquad \square$$

Lemma 5.4. $\mathsf{SD}_c(\mathcal{H}_2, \mathcal{T}, \emptyset)$ *for a tree* $\mathcal{T}$ *of* N *nodes is at most* $\lambda_c(\log N - c) + c + O(1)$*, with properly chosen parameters* $\alpha_{c,i}$*'s.*

Proof. For ease of computation, we define a function D' independent of D, as a recurrence that satisfies the recursion that, by Lemma 5.3, holds for the running time.

$$\begin{cases} D'_{c,i}(N) = D'_{c,1}((1-\alpha_{c,i})N) + 1 = D'_{c,i+1}(\alpha_{c,i}N) + 1 \quad \forall i : 1 \leq i < c\,, \\ D'_{c,c}(N) = D'_{c,1}(N) + 1\,. \end{cases}$$

By manipulating the first equation, we can derive that

$$\begin{aligned} & D'_{c,1}((1-\alpha_{c,i})N) = D'_{c,i+1}(\alpha_{c,i}N) && \forall i : 1 \leq i < c \\ \Rightarrow\quad & D'_{c,1}((1-\alpha_{c,i})/\alpha_{c,i}N) = D'_{c,i+1}(N) && \forall i : 1 \leq i < c \\ \Rightarrow\quad & D'_{c,1}((1-\alpha_{c,i-1}/\alpha_{c,i-1}N) = D'_{c,i}(N) && \forall i : 1 < i \leq c\,. \end{aligned}$$

Again, by the first equation, for each $1 \leq i < c$,

$$D'_{c,i}(N) = D'_{c,i+1}(\alpha_{c,i}N) + 1 = D'_{c,1}(\alpha_{c,i}(1-\alpha_{c,i+1})N) + 2\,,$$

thus

$$D'_{c,1}(N) = D'_{c,1}(\alpha_{c,i}(1-\alpha_{c,i+1})/(1-\alpha_{c,i})N) + 2 \qquad \forall i : 1 \leq i < c\,.$$

Consider minimizing the values of $D'_{c,1}$. Since $D'_{c,1}(N) = D'_{c,1}((1-\alpha_{c,1})N) + 1$, we let

$$1 - \alpha_{c,1} = \alpha_{c,i}\frac{1-\alpha_{c,i+1}}{1-\alpha_{c,i}}\,.$$

By rearranging,

$$\alpha_{c,i+1} = 1 - \frac{(1-\alpha_{c,1})(1-\alpha_{c,i})}{\alpha_{c,i}}$$

holds. Inductively, the following can be proved

$$\alpha_{c,i} = 1 - \frac{\alpha_{c,1}(1-\alpha_{c,1})^i}{2\alpha_{c,1} - 1 + (1-\alpha_{c,1})^i} \qquad \forall i : 1 \le i \le c\,.$$

Now look at the boundary conditions, since

$$D'_{c,c}(N) = D'_{c,1}(N) + 1 = D'_{c,1} \cdot \frac{(1-\alpha_{c,c-1})N}{\alpha_{c,c-1}}\,.$$

Again, to minimize the values of $D'_{c,1}$, let

$$\frac{\alpha_{c,c-1}}{1-\alpha_{c,c}} = 1 - \alpha_{c,1}\,, \qquad \text{and therefore} \qquad \alpha_{c,c-1} = \frac{1-\alpha_{c,1}}{2-\alpha_{c,1}}\,.$$

By what is shown above,

$$\frac{1-\alpha_{c,1}}{2-\alpha_{c,1}} = 1 - \frac{\alpha_{c,1}(1-\alpha_{c,1})^{c-1}}{2\alpha_{c,1} - 1 + (1-\alpha_{c,1})^{c-1}}\,,$$

which implies

$$\sum_{i=1}^{c} (1-\alpha_{c,1})^i = 1\,.$$

We choose $\alpha_{c,1} = \alpha^*_{c,1}$ to be a solution of the equation above, and then all the other $\alpha_{c,i}$'s can be fixed. By setting $\lambda_c = 1/\log(1-\alpha^*_{c,1})$, for each $1 \le i \le c$, $D'_{c,i}(N) \ge D'_{c,i}(N/2) + 1$. We get

$$\begin{cases} D'_{c,i}(N) = \max\{D'_{c,1}((1-\alpha_{c,i})N), D'_{c,i+1}(\alpha_{c,i}N), D'_{c,i}(N/2)\} + 1 \quad \forall i : 1 \le i < c\,, \\ D'_{c,c}(N) = \max\{D'_{c,1}(N), D'_{c,c}(N/2)\} + 1\,. \end{cases}$$

Note that D' is defined in a way that it satisfies the recursion in Lemma 5.3, and it can be proved by induction that $D'_{c,i}(N)$ upper bounds $D_{c,i}(N)$ for all c, i. By the choice of the parameters, the recurrence can be solved using standard tools. Specifically,

$$D_{c,1}(N) \le D'_{c,1}(N) \le \lambda_c(\log N - c) + D_{c,1}(2^c) + O(1)\,.$$

Since $D_{c,1}(2^c) = c$, $\mathsf{SD}_c(\mathcal{H}_c, \mathcal{T}, S)$ is upper bounded by $\lambda_c(\log N - c) + c + O(1)$, where λ_c satisfies

$$\sum_{\ell=1}^{c} 2^{-\frac{\ell}{\lambda_c}} = 1\,. \qquad \square$$

Proof of Theorem 5.1. For every $c \geq 2$, we solve the above recurrences: when $N < 2^c$, the running time is

$$d^{\log N(1-\varepsilon)\mathcal{TW}(\phi)}|\phi|^{O(1)};$$

when $N \geq 2^c$, the running time is

$$d^{(\lambda_c(\log N-c)+c)(1-\varepsilon)\mathcal{TW}(\phi)}|\phi|^{O(1)}.$$

Space required by the algorithm is upper bounded by $2^{c\varepsilon\mathcal{TW}(\phi)}|\phi|^{O(1)}$ since only $type_i$ trees are allowed, for $i \leq c$. □

The value λ_c depending on the choice of parameter c seems quite artificial in the analysis of our algorithms, but later we will see that this constant is actually tight. Here is an upper bound on λ_c.

Lemma 5.5. $\lambda_c < 1 + \dfrac{2}{2^{c/2}}$.

Proof. Let $f(X) = X^c - \sum_{i=0}^{c-1} X^i$, and let γ_c be the root of $f(X) = 0$ with largest absolute value. We know $f(2) = 1 > 0$, so if we can prove $f(2-1/2^{c/2}) < 0$ then there must be a root between 2 and $2-1/2^{c/2}$. Denote $y = 2-1/2^{c/2}$,

$$f(y) < 0 \iff y^c < \sum_{i=0}^{c-1} y^i = \frac{y^c-1}{y-1} \iff y < 2 - \frac{1}{y^c}.$$

The last inequality is true because $y = 2-1/2^{c/2} > \sqrt{2}$ when $c \geq 2$ and $2-1/y^c > 2-1/\sqrt{2}^c = y$. By definition of $\lambda_c = 1/\log_2 \gamma_c$, it follows that $\lambda_c < 1+2/2^{c/2}$. □

Given the above upper bound, we can furthermore prove an interesting feature of our family of algorithms. Namely, the space resource can be fully exploited to minimize the running time, which potentially is of practical importance.

Corollary 5.6 (of Theorem 5.1). *For any $\varepsilon' > 0$ there exists an algorithm which runs in space $2^{\varepsilon'\mathcal{TW}(\phi)}|\phi|^{O(1)}$ and time $d^{\delta\mathcal{TW}(\phi)\log_2|\phi|}|\phi|^{O(1)}$ for a constant $\delta < 1$.*

Proof. For any fixed ε and c, there is an algorithm with running time

$$O\left(d^{\lambda_c(1-\varepsilon)\mathcal{TW}(\phi)\log|\phi|}|\phi|^{O(1)}\right)$$

and space $O(2^{c\varepsilon\mathcal{TW}(\phi)}|\phi|^{O(1)})$ by Theorem 5.1. Set $\varepsilon = \varepsilon'/c$, then the space is $O(2^{\varepsilon'\mathcal{TW}(\phi)}|\phi|^{O(1)})$ and the running time is

$$O\left(d^{\lambda_c(1-\frac{\varepsilon'}{c})\mathcal{TW}(\phi)\log|\phi|}|\phi|^{O(1)}\right).$$

By Lemma 5.5,

$$\lambda_c\left(1-\frac{\varepsilon'}{c}\right) < \left(1+\frac{2}{2^{c/2}}\right)\left(1-\frac{\varepsilon'}{c}\right) < 1$$

for sufficiently large c. □

5.2 Optimality of the generalized tradeoff algorithm

Similarly to the last part of Section 4, we also prove the optimality of the generalized tradeoff algorithm. However, in this case the matching lower bound is more involved (since the upper bound involved a lot of guessing). We construct the hard instance using extended generalized Fibonacci trees.

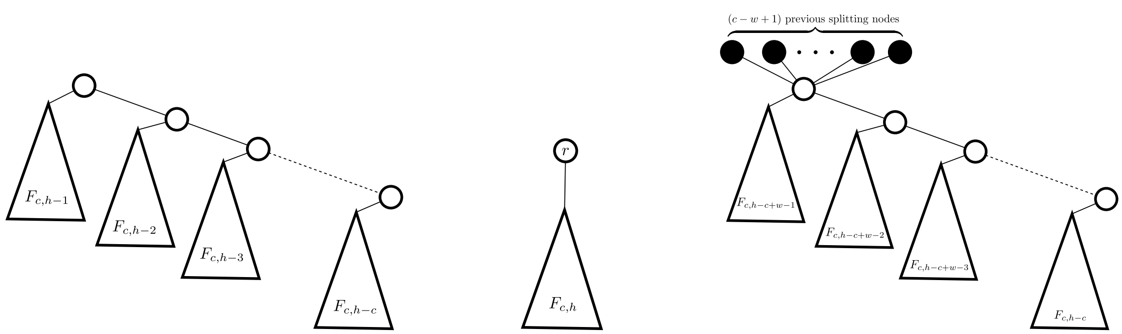
Definition 5.7. For any integer $c \geq 2$, and positive integer h, a (c,h)-*Fibonacci tree*(denoted as $F_{c,h}$) is a rooted tree defined by one of the rules,

(1) if $h \leq c$, $F_{c,h}$ is a chain of 2^c nodes;

(2) if $h > c$, $F_{c,h}$ is constructed by starting from a chain of c nodes (one end as the root), then replacing the ith node (starting from the root) by a subtree $F_{c,h-i}$.

An *extended* (c,h,r)-*Fibonacci tree* (denote as $F^*_{c,h,r}$) is constructed by connecting one root node r to a subtree $F_{c,h}$.

See Figure 9a for an illustration of a (c,h)-Fibonacci tree. $F_{c,h+c}$ is indeed the hard instance for splitting algorithms in $\mathfrak{A}_c$ for $c \geq 2$. As discussed previously, it suffices to lower bound minSD on this tree.



(a) $F_{c,h}$ (b) $F^*_{c,h,r}$ (c) $G_{c,h,w}$

Figure 9: Fibonacci tree in a more general form and the hard instance based on it.

Similar to the proof of Theorem 4.12, we need a definition of trees which may appear in intermediate phases and contain previous splitting nodes. $\forall c \geq 2$, $\forall h > c$ and $\forall w : 1 \leq w \leq c$, $G_{c,h,w}$ is such a tree: first construct a chain of length w, then connect $c-w+1$ previous splitting nodes to the first node of the chain, and $\forall i : 1 \leq i \leq w$, connect a subtree $F_{c,h-c+w-i}$ to the i-th node of the chain. Denote S_ℓ as the set of the ℓ previous splitting nodes connected to the first node of the chain. The following bound holds for these intermediate trees.

Lemma 5.8. $\forall h \geq 1, \forall w : 1 \leq w \leq c$, $\mathsf{minSD}(G_{c,h,w}, S_{c-w+1}) \geq h - c + w$.

Proof. We prove the lemma by induction on h. The base case is trivial. Suppose $\forall h : h < h_0$,

$$\mathsf{minSD}_c(G_{c,h,w}, S_{c-w+1}) \geq h - c + w$$

holds for any valid w. For the induction step, we prove

$$\mathsf{minSD}_c(G_{c,h_0,w}, S_{c-w+1}) \geq h_0 - c + w \tag{5.1}$$

holds for any w by induction on w. When $w = 1$, to prevent $type_i$ trees for $i > c$, we must split at the first node of the chain. By induction hypothesis (on h),

$$\mathsf{minSD}_c(G_{c,h_0,1}, S_c) \geq 1 + \mathsf{minSD}_c(G_{c,h_0-c,c}, S_1) \geq h_0 - c + 1\,.$$

Suppose (5.1) holds $\forall w : w < w_0$, for the induction step, consider lower bounding

$$\mathsf{minSD}_c(G_{c,h_0,w_0}, S_{c-w_0+1})\,.$$

As in the proof of Lemma 4.13, in order to apply the induction hypothesis, some size-reducing procedure which will not increase the running time will be applied to the tree after splitting. If the splitting node is inside the subtree F_{c,h_0-c+w_0-1}, prune the nodes in the subtree F_{c,h_0-c+w_0-1} outside the path between the first node on the chain and the splitting node, then shrink the path to an edge, and finally shrink the first edge on the chain to obtain a G_{c,h_0,w_0-1}, by induction hypothesis,

$$\mathsf{minSD}_c(G_{c,h_0,w_0-1}, S_{c-(w_0-1)+1}) \geq h_0 - c + w_0 - 1\,;$$

otherwise consider an additional free splitting at the first node on the chain, this produces a $G_{c,h_0-c+w_0-1,c}$, again by induction hypothesis ($h_0 - c + w_0 - 1 < h_0$),

$$\mathsf{minSD}_c(G_{c,h_0-c+w_0-1,c}, S_1) = h_0 - c + w_0 - 1 - c + c = h_0 - c + w_0 - 1\,.$$

Therefore,

$$\mathsf{minSD}_c(G_{c,h_0,w_0}, S_{c-w_0+1}) \geq 1 + h_0 - c + w_0 - 1 = h_0 - c + w_0\,.$$

This completes both inductions and also the proof. □

Corollary 5.9. *For each $h \geq 1$, $\mathsf{minSD}(F^*_{c,h,r}, \{r\}) \geq h$.*

Proof. By Lemma 5.8, $\mathsf{minSD}_c(F^*_{c,h,r}, \{r\}) = \mathsf{minSD}_c(G_{c,h,c}, S_1) \geq h - c + c \geq h$. □

Now we are ready to state the theorem for optimality.

Theorem 5.10. *For every $c \geq 2$ and $N > 2^c$, there exists a tree $\mathcal{T}$ with N nodes, such that*

$$\mathsf{minSD}_c(\mathcal{T}, \emptyset) \geq \lambda_c(\log N - c) + c - O(1)\,.$$

Proof. Let $|F_{c,h}|$ be the number of nodes in the tree $F_{c,h}$. For any $h \leq c$, we have $|F_{c,h}| \leq 2^c$, when $h > c$, we have $|F_{c,h}| = \sum_{i=1}^{c} |F_{c,h-i}| + c$. By the recursion, the generating function of $|F_{c,h}|$ can be written as $f(X) = X^c - \sum_{i=0}^{c} X^i$. Therefore

$$|F_{c,h}| = \sum_{i=1}^{c} \delta_{c,i} \gamma_{c,i}^{h-c}\,,$$

where $\delta_{c,i}$ is upper bounded by a constant times 2^c and $\gamma_{c,i}$ is the i-th root of the equation $f(X) = 0$.

Let $\gamma_c = \arg\max_i\{|\gamma_{c,i}|\}$. When h tends to infinity, $|F_{c,h}| = \Theta(2^c \gamma_c^{h-c})$. So,

$$h \geq \log_{\gamma_c}(|F_{c,h}|/2^c) + c - O(1) = \lambda_c(\log|F_{c,h}| - c) + c - O(1)\,.$$

Therefore, for any $c \geq 2$ and large enough N, consider the tree $\mathcal{T} = F_{c,h+c}$ with $\geq N$ nodes. Splitting it at any node will result in a tree, whose minimum splitting depth is lower bounded by $\mathsf{minSD}(F^*_{c,h,r}, \{r\})$: if the splitting node is on the chain, then let r be the splitting node; otherwise let r be any node on the chain other than the one which is connected to the root of the subtree where the splitting node lay. (One can think of as splitting at r for free.) This, when combining with Corollary 5.9, completes the proof. □

Similarly to Theorem 4.12, here we conclude the optimality of our tradeoff algorithm. That is, for fixed $c \geq 2$, $\forall \varepsilon : 0 < \varepsilon < 1$, and every sufficiently large input length and every algorithm $\mathfrak{A}_c$ (whose description may depend on the length) there is an instance ϕ, for which the running time is

$$\Omega\big(3^{\lambda_c(\log|\phi| - c) + c - O(1)}|\phi|^{\Theta(1)}\big)\,.$$

6 Future work

A very exciting research direction is to unconditionally verify our conjecture in restricted models of computation—propositional proof complexity lower bounds can be understood as such results. The work of Beame-Beck-Impagliazzo [6] took the first step towards this direction. Such results can be also understood as partial progress towards $\mathsf{SC} \neq \mathsf{NC}$.

A rather intriguing direction regarding positive results, is to use randomness in order to improve the multiplicative constants in the exponents of time or space, or to provide improved tradeoffs. More generally, we would like to understand the role of randomness in width-parameterized SAT-solving, a topic which is fundamentally unexplored.

Acknowledgments

We would like to thank Hubie Chen, Kevin Matulef and Alexander Razborov for useful remarks and suggestions.

References

[1] MANINDRA AGRAWAL, ERIC ALLENDER, SAMIR DATTA, HERIBERT VOLLMER, AND KLAUS W. WAGNER: Characterizing small depth and small space classes by operators of higher type. *Chicago J. Theor. Comput. Sci*, 2000(2), 2000. CJTCS. 302

[2] MICHAEL ALEKHNOVICH AND ALEXANDER A. RAZBOROV: Satisfiability, branch-width and Tseitin tautologies. In *Proc. 43rd FOCS*, pp. 593–603. IEEE Comp. Soc. Press, 2002. See also journal version in Comput. Complexity 20(4), 2011, 649-678. [doi:10.1109/SFCS.2002.1181983] 298, 299, 301

[3] SANJEEV ARORA AND BOAZ BARAK: *Computational complexity: a modern approach*. Volume 1. Cambridge Univ. Press, 2009. [doi:10.1017/CBO9780511804090] 312

[4] FAHIEM BACCHUS, SHANNON DALMAO, AND TONIANN PITASSI: Solving #SAT and Bayesian inference with backtracking search. *J. Artif. Intell. Res. (JAIR)*, 34:391–442, 2009. Preliminary version in FOCS'03. [doi:10.1613/jair.2648] 299, 301, 312

[5] DAVID A. MIX BARRINGTON, NEIL IMMERMAN, AND HOWARD STRAUBING: On uniformity within NC^1. *J. Comput. System Sci.*, 41(3):274–306, 1990. [doi:10.1016/0022-0000(90)90022-D] 306

[6] PAUL BEAME, CHRISTOPHER BECK, AND RUSSELL IMPAGLIAZZO: Time-space tradeoffs in resolution: superpolynomial lower bounds for superlinear space. In *Proc. 44th STOC*, pp. 213–232. ACM Press, 2012. See also at ECCC. [doi:10.1145/2213977.2213999] 302, 313, 334

[7] HANS L. BODLAENDER: A tourist guide through treewidth. *Acta Cybern.*, 11(1-2):1–22, 1993. See at Act Cybernetica. [http:citeseerx.ist.psu.eduviewdocsummary?doi=10.1.1.18.8755] 300, 314

[8] HANS L. BODLAENDER: A partial *k*-arboretum of graphs with bounded treewidth. *Theoret. Comput. Sci.*, 209(1-2):1–45, 1998. [doi:10.1016/S0304-3975(97)00228-4] 303, 315

[9] HANS L. BODLAENDER, FEDOR V. FOMIN, ARIE M. C. A. KOSTER, DIETER KRATSCH, AND DIMITRIOS M. THILIKOS: A note on exact algorithms for vertex ordering problems on graphs. *Theory of Computing Systems*, 50(3):420–432, 2012. [doi:10.1007/s00224-011-9312-0] 301

[10] HANS L. BODLAENDER, JOHN R. GILBERT, HJÁLMTÝR HAFSTEINSSON, AND TON KLOKS: Approximating treewidth, pathwidth, frontsize, and shortest elimination tree. *J. Algorithms*, 18(2):238–255, 1995. [doi:10.1006/jagm.1995.1009] 309

[11] RONALD V. BOOK, CHRISTOPHER B. WILSON, AND XU MEI-RUI: Relativizing time, space, and time-space. *SIAM J. Comput.*, 11(3):571–581, 1982. [doi:10.1137/0211048] 313

[12] ALLAN BORODIN, STEPHEN A. COOK, PATRICK W. DYMOND, WALTER L. RUZZO, AND MARTIN TOMPA: Two applications of inductive counting for complementation problems. *SIAM J. Comput.*, 18(3):559–578, 1989. Preliminary version in SCTC'88 Erratum in SICOMP. [doi:10.1137/0218038] 306

[13] STEPHEN A. COOK: Characterizations of pushdown machines in terms of time-bounded computers. *J. ACM*, 18(1):4–18, 1971. Preliminary version in STOC'69. [doi:10.1145/321623.321625] 306

[14] STEPHEN A. COOK: A taxonomy of problems with fast parallel algorithms. *Inf. Control*, 64(1-3):2–22, 1985. Preliminary version in FCT'83. [doi:10.1016/S0019-9958(85)80041-3] 302

[15] STEPHEN A. COOK AND ROBERT A. RECKHOW: The relative efficiency of propositional proof systems. *J. Symbolic Logic*, 44(1):36–50, 1979. Preliminary version in STOC'74. [doi:10.2307/2273702] 302

[16] ELDAR FISCHER, JOHANN A. MAKOWSKY, AND ELENA V. RAVVE: Counting truth assignments of formulas of bounded tree-width or clique-width. *Discr. Appl. Math.*, 156(4):511–529, 2008. [doi:10.1016/j.dam.2006.06.020] 301

[17] FEDOR V. FOMIN, FABRIZIO GRANDONI, DANIEL LOKSHTANOV, AND SAKET SAURABH: Sharp separation and applications to exact and parameterized algorithms. *Algorithmica*, 63(3):692–706, 2012. Preliminary version in LATIN'10. [doi:10.1007/s00453-011-9555-9] 302

[18] FEDOR V. FOMIN AND DIETER KRATSCH: *Exact Exponential Algorithms*. Springer, 2010. [doi:10.1007/978-3-642-16533-7] 301

[19] KONSTANTINOS GEORGIOU AND PERIKLIS A. PAPAKONSTANTINOU: Complexity and algorithms for well-structured k-SAT instances. In *Theory and Applications of Satisfiability Testing - SAT 2008*, volume 4996 of *LNCS*, pp. 105–118. Springer, 2008. [doi:10.1007/978-3-540-79719-7_10] 299, 301, 310, 313

[20] GEORG GOTTLOB, NICOLA LEONE, AND FRANCESCO SCARCELLO: The complexity of acyclic conjunctive queries. *J. ACM*, 48(3):431–498, 2001. Preliminary version in FOCS'98. [doi:10.1145/382780.382783] 299, 311

[21] MARTIN GROHE: The complexity of homomorphism and constraint satisfaction problems seen from the other side. *J. ACM*, 54(1):1–24, 2007. Preliminary version in FOCS'03. [doi:10.1145/1206035.1206036] 302

[22] RUSSELL IMPAGLIAZZO, RAMAMOHAN PATURI, AND FRANCIS ZANE: Which problems have strongly exponential complexity? *J. Comput. Syst. Sci.*, 63(4):512–530, 2001. Preliminary version in FOCS'98. [doi:10.1006/jcss.2001.1774] 302

[23] TON KLOKS: *Treewidth, Computations and Approximations*. Volume 842 of *Lecture Notes in Computer Science*. Springer, 1994. [doi:10.1007/BFb0045375] 303

[24] MIKKO KOIVISTO AND PEKKA PARVIAINEN: A space-time tradeoff for permutation problems. In *Proc. 21st Ann. ACM-SIAM Symp. on Discrete Algorithms (SODA'10)*, pp. 484–492. ACM Press, 2010. [doi:10.1137/1.9781611973075.41] 301

[25] DANIEL LOKSHTANOV, DÁNIEL MARX, AND SAKET SAURABH: Known algorithms on graphs on bounded treewidth are probably optimal. In *Proc. 22nd Ann. ACM-SIAM Symp. on Discrete Algorithms (SODA'11)*, pp. 777–789, 2011. [doi:10.1137/1.9781611973082.61, arXiv:1007.5450] 302

[26] DANIEL LOKSHTANOV, MATTHIAS MNICH, AND SAKET SAURABH: Planar k-path in subexponential time and polynomial space. In *Graph-Theoretic Concepts in Computer Science - 37th International Workshop, WG 2011*, pp. 262–270, 2011. [doi:10.1007/978-3-642-25870-1_24] 301

[27] DÁNIEL MARX: Can you beat treewidth? *Theory of Computing*, 6(5):85–112, 2010. Preliminary version in FOCS'07. [doi:10.4086/toc.2010.v006a005] 302

[28] ROBIN A. MOSER AND DOMINIK SCHEDER: A full derandomization of Schöning's k-SAT algorithm. In *Proc. 43rd STOC*, pp. 245–252. ACM Press, 2011. [doi:10.1145/1993636.1993670, arXiv:1008.4067] 301

[29] ROLF NIEDERMEIER AND PETER ROSSMANITH: Unambiguous auxiliary pushdown automata and semi-unbounded fan-in circuits. *Inform. and Comput.*, 118(2):227–245, 1995. Preliminary version in LATIN'92. [doi:10.1006/inco.1995.1064] 307

[30] PERIKLIS A. PAPAKONSTANTINOU: A note on width-parameterized SAT: An exact machine-model characterization. *Inform. Process. Lett.*, 110(1):8–12, 2009. [doi:10.1016/j.ipl.2009.09.012] 302, 306, 310

[31] RAMAMOHAN PATURI, PAVEL PUDLÁK, MICHAEL E. SAKS, AND FRANCIS ZANE: An improved exponential-time algorithm for k-SAT. *J. ACM*, 52(3):337–364, 1998. Preliminary version in FOCS'98. [doi:10.1145/1066100.1066101] 301

[32] NEIL ROBERTSON AND PAUL D. SEYMOUR: Graph minors. I. excluding a forest. *J. Comb. Theory, Ser. B*, 35(1):39–61, 1983. [doi:10.1016/0095-8956(83)90079-5] 300

[33] NEIL ROBERTSON AND PAUL D. SEYMOUR: Graph minors. II. algorithmic aspects of tree-width. *J. Algorithms*, 7(3):309–322, 1986. [doi:10.1016/0196-6774(86)90023-4] 300

[34] WALTER L. RUZZO: Tree-size bounded alternation. *J. Comput. System Sci.*, 21(2):218–235, 1980. Preliminary version in STOC'79. [doi:10.1016/0022-0000(80)90036-7] 305, 306

[35] MARKO SAMER AND STEFAN SZEIDER: Algorithms for propositional model counting. *J. Discrete Algorithms*, 8(1):50–64, 2010. Preliminary version in LPAR'07. [doi:10.1016/j.jda.2009.06.002] 300, 301, 313

[36] UWE SCHÖNING: A probabilistic algorithm for k-SAT based on limited local search and restart. *Algorithmica*, 32(4):615–623, 2002. Preliminary version in FOCS'99. [doi:10.1007/s00453-001-0094-7] 301

[37] STEFAN SZEIDER: On fixed-parameter tractable parameterizations of SAT. In *Theory and Applications of Satisfiability Testing - SAT 2003*, volume 2919 of *LNCS*, pp. 188–202, 2003. [doi:10.1007/978-3-540-24605-3_15] 301

[38] H. VENKATESWARAN: Properties that characterize LOGCFL. *J. Comput. Syst. Sci.*, 43(2):380–404, 1991. Conference version in STOC'87. [doi:10.1016/0022-0000(91)90020-6] 305, 306, 307

[39] HERIBERT VOLLMER: *Introduction to Circuit Complexity*. Springer, 1999. [doi:10.1007/978-3-662-03927-4] 306, 307

[40] GERHARD J. WOEGINGER: Exact algorithms for NP-hard problems: A survey. *Combinatorial Optimization—Eureka, You Shrink!*, 2570:185–207, 2003. [doi:10.1007/3-540-36478-1_17] 301

AUTHORS

Eric Allender
Distinguished professor
Rutgers University, New Brunswick, NJ
allender@cs.rutgers.edu
http://www.cs.rutgers.edu/~allender

Shiteng Chen
Ph. D. student
Institute for Interdisciplinary Information Sciences
Tsinghua University
shitengchen@gmail.com
http://iiis.tsinghua.edu.cn/shitengchen

Tiancheng Lou
Software engineer
Google Mountain View
tiancheng.lou@gmail.com
http://iiis.tsinghua.edu.cn/tianchenglou

Periklis A. Papakonstantinou
Assistant professor
Institute for Interdisciplinary Information Sciences
Tsinghua University
papakons@tsinghua.edu.cn
http://iiis.tsinghua.edu.cn/~papakons

Bangsheng Tang
Associate researcher
Hulu, Beijing Office
bangsheng.tang@gmail.com
http://tang.bangsheng.info

ABOUT THE AUTHORS

ERIC ALLENDER is a professor at Rutgers University. He has been at Rutgers since receiving his Ph. D. in 1985 at Georgia Tech, under the supervision of Kim N. King. While at Georgia Tech, he was the Backbone of the Seed and Feed Marching Abominable and he still plays trombone from time to time. He did his undergraduate work at the University of Iowa. He is a Fellow of the ACM, and currently serves as Editor-in-Chief of ACM Transactions on Computation Theory. Circuit complexity, Kolmogorov complexity, and complexity classes are his main research interests. He and his wife find happiness on the dance floor and toiling in their garden.

SHITENG CHEN is currently a Ph. D. student at the Institute for Interdisciplinary Information Sciences, Tsinghua University, advised by Periklis Papakonstantinou. He did his undergraduate studies in the pilot computer science program at Tsinghua University, the "Yao class."

TIANCHENG LOU is currently with Google. He received his Ph. D. in 2012 from the Institute for Interdisciplinary Information Sciences, Tsinghua University under the supervision of Professor Andrew C. Yao. He holds numerous programming contest awards.

PERIKLIS A. PAPAKONSTANTINOU is an assistant professor at the Institute for Interdisciplinary Information Sciences, Tsinghua University. He took up this appointment (in 2010) immediately after some wonderful years at the University of Toronto, where he had the privilege of being part of a great theory group. During his Ph. D. at the University of Toronto he read theory of computing and cryptography with his advisor Charlie Rackoff, whose teachings contributed to his deep appreciation for concepts in Computer Science. Significant influence on his research identity and taste comes from his teachers Gábor Pete and Balázs Szegedy (from studies in mathematics at the University of Toronto), and Stavros Cosmadakis (from his engineering studies at the University of Patras in Greece). His current research interests are in the foundations of computer science (at large), and he has a more than occasional interest in Cryptography and Machine Learning. In his early youth he enjoyed riding motorbikes and collecting graduate degrees.

BANGSHENG TANG is currently an Associate Researcher at Hulu, Beijing Office. He received his Ph. D. in 2013 from the Institute for Interdisciplinary Information Sciences, Tsinghua University, under the supervision of Periklis A. Papakonstantinou. This work was done during his Ph. D. studies. He did his undergraduate studies in the first-ever Tsinghua University Special Pilot CS Class, founded by Andrew C. Yao. His research interests include algorithm design, computational complexity and computational group theory.