Electronic Colloquium on Computational Complexity, Revision 1 of Report No. 27 (2012)

Width-parameterized SAT: time-space tradeoffs

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

*Department of Computer Science, Rutgers University
†Institute for Theoretical Computer Science, Tsinghua University

March 29, 2012

Abstract

A decade has passed since Alekhnovich and Razborov [AR02] presented an algorithm that solves **SAT** on instances ϕ of size n having tree-width $\mathcal{TW}(\phi)$, using time (and space) bounded by $2^{O(\mathcal{TW}(\phi))}n^{O(1)}$. Although there have been several papers over the ensuing years building on the work of Alekhnovich and Razborov (e.g. [BDP09, BL03, FMR08, SS10]) there has been no real progress on what is listed as the first open question of [AR02]: Can one "do anything intelligent in polynomial space to check satisfiability of formulas" with small tree-width? We present both positive and negative results on this question; we present a fairly fast polynomial space algorithm, and present complexity-theoretic evidence that no significantly faster algorithm runs in polynomial space.

Our first positive result is a simple algorithm that runs in polynomial space and achieves run-time $3^{\mathcal{TW}(\phi)\log n}n^{O(1)}$, nearly matching the run-time of [AR02], but with an annoying factor of $\log n$ in the exponent.

Our negative results indicate that this annoying factor of $\log n$ is unavoidable. For ease of exposition, let us focus on the case where the tree-width is $\log^k n$. Then, when $k = 1$ we show that solving **SAT** instances of tree-width $\log n$ is complete for $\mathbf{LOGCFL} = \mathbf{SAC}^1$, and for arbitrary k, **SAT** of tree-width $\log^k n$ is complete for a level of the **NSC** hierarchy corresponding to log-depth semi-unbounded fan-in circuits of size $2^{O(\log^k n)}$. (**NSC** is the class of problems solvable in nondeterministic polynomial time and poly-logarithmic space: i.e., the nondeterministic analog of the well-known class **SC**.) Problems in this class can be solved in space $\log^{k+1} n$ (and hence in time $2^{O(\log^{k+1} n)}$), and also in time $2^{O(\log^k n)}$ (with space bound the same as the time bound). These results show that our conjecture (that the annoying factor of $\log n$ in the exponent of the running time of our polynomial-space algorithm cannot be eliminated) 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, possibly with different resource bounds depending on k, of the long-standing conjecture in complexity theory that $\mathbf{SAC}^1$ (and even its subclass **NL**) is not contained in **SC**, or even in the Time-Space class $\mathbf{TISP}(n^{O(1)}, 2^{\log^{1-\varepsilon} n})$.

The most involved part of this paper is the demonstration that the best-known time-efficient and space-efficient algorithms for small tree-width **SAT** can be combined using a new technique to obtain, for each ε with $0 < \varepsilon < 1$, an algorithm with time-space complexity $\left(3^{1.441(1-\varepsilon)\mathcal{TW}(\phi)\log|\phi|}|\phi|^{O(1)},\ 2^{2\varepsilon\mathcal{TW}(\phi)}|\phi|^{O(1)}\right)$. 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.

ISSN 1433-8092

1 Introduction

In this paper we focus on the prototypical **NP**-complete problem **SAT**, where problem instances have small *tree-width*.[1] This restriction of **SAT** was studied by Alekhnovich and Razborov [AR02] (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 is the number variables and clauses.

The motivation for studying this restriction is that real-world **SAT** instances frequently tend to have small width. The authors of [AR02] 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. It is well-known in the SAT-solving community that in many common cases SAT-solvers abort due to lack of space.

We provide answers to this question of Alekhnovich and Razborov.

First, 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 of arbitrary CNF instances, that runs in polynomial space.[2]

Our simple algorithm does not fully answer the question of [AR02], since we suffer a $\log|\phi|$ factor in the exponent of the running time. Most of our work revolves around this logarithmic factor. We conjecture that it cannot be removed:

Conjecture 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 $\mathcal{TW}(\phi) = O(|\phi|^{1-\epsilon})$, for some fixed $\epsilon < 0$. If $\delta(\phi) = o(\log|\phi|)$ then $\mathcal{A}$ uses space $2^{\Omega(\mathcal{TW}(\phi))}$.*

We offer complexity-theoretic evidence in support of this conjecture. We show that solving **SAT** instances of small tree-width constitutes a complete problem for a subclass of **NP** defined in terms of a well-studied type of Boolean circuits: semi-unbounded circuits of logarithmic depth. Semi-unbounded fan-in circuits provide one of the standard characterizations of the class $\mathbf{SAC}^1$, which is not believed to lie in $\mathbf{TISP}(n^{O(1)}, n^{o(1)})$. This can be seen as a strong form of the popular conjecture $\mathbf{NC} \not\subseteq \mathbf{SC}$. Our study shows that Conjecture 1 can be seen as simply a more general form of this belief, for a wider range of resource bounds. This connection is explored in more detail in Section 3.

Assuming for now that Conjecture 1 holds (which implies that there are severe limits to how much time complexity can be improved without sacrificing space complexity), it makes sense to devise algorithms that approach these limits. This is the topic of Section 4, which constitutes the most technically-involved part of this work, though the practical significance of these algorithms is debatable.

[1] Definitions of formula tree-width and related notions are deferred until Section 2.

[2] The polynomial space algorithm of [GP08] works only for k-CNFs, whereas the one of [BDP09] is for tree-width on the primal graph.

1.1 Related work

Tree-width is a popular graph parameter introduced by Robertson and Seymour [RS83, RS86]. 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. [Bod93] 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 upper bounds the tree-width value of the rest [Sze04]. 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. [BDP09]. 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.

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. [FK10, Woe03] 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. [MS11, PPSZ98, Sch99, Woe03]. 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 [BFK$^+$12] and the references within (in particular [KP10]). However, there is no easy way to adapt these techniques in our setting, and if Conjecture 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.

Algorithms for width-parameterized SAT. Prior to our work, [BDP09, GP08] addressed the question of Alekhnovich and Razborov. In [GP08] 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 problem is not only that their algorithm works only for k-**SAT**, but also due to the non-explicitness the constant in the exponent of the running time cannot be bounded in any easy way. [BDP09] presents 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 [AR02], considering different width-parameters: Fischer et al. [FMR08] 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 [SS10], which we make use of later in this paper as a time-efficient algorithm, running in time-space $\left(2^{2\mathcal{TW}(\phi)}|\phi|^{O(1)},\ 2^{\mathcal{TW}(\phi)}|\phi|^{O(1)}\right)$.

We should also mention that in two excellent works on constraint satisfaction problems Grohe

[Gro07] and Marx [Mar10] essentially show that the running time of the known width-based algorithms is optimal.

Previous hardness results, and NSC characterizations. [Pap09] considers path-width instead of tree-width, and shows that the complexity of deciding path-width parameterized **SAT** instances precisely corresponds to the streaming verification in log-space of **NP**-witnesses. In particular, [Pap09] shows that deciding formulas with given path decompositions of width $O(\log|\phi|)$ is complete for **NL**, and asks whether the complexity of **SAT** instances with tree-width $O(\log|\phi|)$ is more difficult. In this paper we answer this question affirmatively, unless $\mathbf{NL} = \mathbf{SAC}^1$. (It is conjectured, e.g. [Coo85], that $\mathbf{SAC}^1 = \mathbf{LOGCFL} \neq \mathbf{NL}$.) We characterize these "streaming verification classes" by levels of **NSC**. Then, we give a natural circuit characterization of intermediate levels in the **NSC** hierarchy using semi-unbounded-fan-in circuits of depth $O(\log n)$ and size quasi-polynomial. For these classes we obtain new hardness results. Let us note that this characterization of **NSC** is of independent interest, and is perhaps more natural than the characterizations presented in [AAD+00].

Relation to Propositional Proof Complexity. Our work opens new, exciting directions for Propositional Proof Complexity. One way to make progress towards validating Conjecture 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 very recently [BBI11] made progress towards *exactly* validating our question. In particular, they proved a Resolution Refutation size-space tradeoff, which in particular 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 exciting development is the first super-polynomial lower bound and through our work it can be interpreted as validating the $\mathbf{SAC}^1 \not\subseteq \mathbf{SC}$ conjecture, at least for a class of restricted algorithms. This is a new direction; lower bounds in proof complexity are clearly connected to the $\mathbf{NP} \neq \mathbf{coNP}$ conjecture [CR79], but have not previously seemed to have a bearing on the $\mathbf{SC} \neq \mathbf{NC}$ question.

1.2 Our contribution and techniques

In Section 3 we show that **SAT** of a given tree-decomposition of width $\log^k n$ is complete for the class $\mathbf{SAC}^k_{\mathsf{quasi}} := \mathbf{SAC}(O(\log n), 2^{O(\log^k n)})$, 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 [GLS01]. We also show that $\mathbf{NSC}^k = \mathbf{NTISP}(n^{O(1)}, O(\log^k n))$ is contained in $\mathbf{SAC}^k_{\mathsf{quasi}}$ which is in turn contained in $\mathbf{NSC}^{k+1}$, where **NTISP** denotes the set of problems decidable by non-deterministic Turing Machines that are simultaneously TIme-SPace bounded. Note that the **NSC** levels are direct space-scaled analogs of **NL** and these **SAC** classes are direct size-scaled analogs of $\mathbf{SAC}^1$. 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 $\mathbf{SAC}^1$.

On the positive side, we give a recursive algorithm that runs in time-space $\left(3^{\mathcal{TW}(\phi)\log|\phi|}|\phi|^{O(1)}, |\phi|^{O(1)}\right)$ (see Section 4). This is the first space-efficient algorithm for width-parameterized **SAT** for arbitrary CNFs.

We use this space-efficient algorithm, together with a time-efficient dynamic programming algorithm (essentially the algorithm of [SS10]), 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 infinite pairs of values are of interest, depending on the different time-space bounds one may want to achieve.

Note: 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 work in width-parameterized **SAT**.

2 Preliminaries

We introduce notation, terminology, and conventions used throughout the paper. We also provide a rather elementary introduction on how an algorithm may exploit the structure of bounded tree-width formulas.

2.1 Notation

All logarithms are of base 2, and all propositional formulas are in Conjuctive Normal Form (CNF). **SAT** is the decision problem where given an arbitrary CNF formula we want to decide if it is satisfiable. k-**SAT** denotes 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 1. *Let $G = (V, E)$ be an undirected graph. A* tree decomposition *of G is a tuple (T, X), where $T = (W, F)$ is a tree, and $X = \{X_1, \cdots, X_{|W|}\}$ where $X_i \subseteq V$ s.t.*

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

(2) $\forall (i, j) \in E$, $\exists t \in W$, *s.t.* $i, j \in X_t$.

(3) $\forall i$, *the set* $\{t : i \in X_t\}$ *forms a subtree of* 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 $T = (W, F)$ is restricted to a path, the decomposition is called *path decomposition*, and the specific tree-width is called path-width $\mathcal{PW}(G)$. The following inequality

holds (e.g. [Bod98])

$$\mathcal{TW}(G) \leq \mathcal{PW}(G) \leq O(\log |V| \mathcal{TW}(G)) \tag{1}$$

Definition 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 unsigned 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 the input tree decompositions have the following two properties.

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

(2) The tree T has bounded degree 3.

Tree decompositions satisfying the two properties are called *nice*. A tree decomposition can be converted to a nice one in linear time (see e.g. [Klo94, Bod98]). 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 1. *We present our results with the parameter* d*. One may replace* d *by* 2 *or* 3 *when the structure of the input decomposition is a path or a tree.*

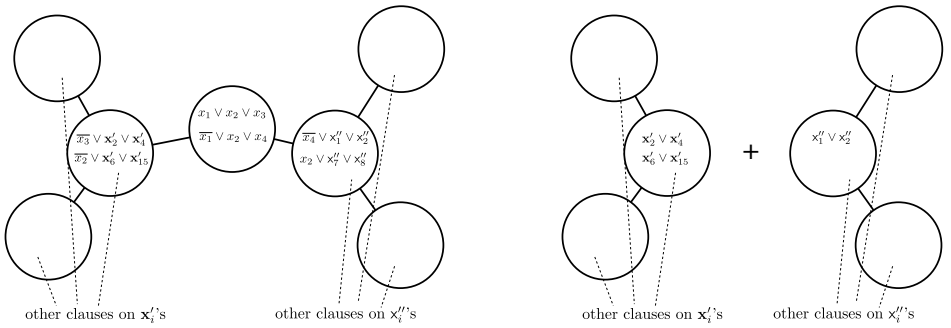
2.3 A primer to algorithms for width-parameterized SAT

The structure of a tree decomposition is associated with the concept of separability(e.g. [Bod98]). 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. For the sake of simplicity we make an additional assumption on the tree decompositions given in the input 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).

Suppose x_i's, x'_i's and x''_i's are different sets of variables and the tree decomposition is as in Figure 1a.

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 1b. 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



(a) Input tree decomposition.

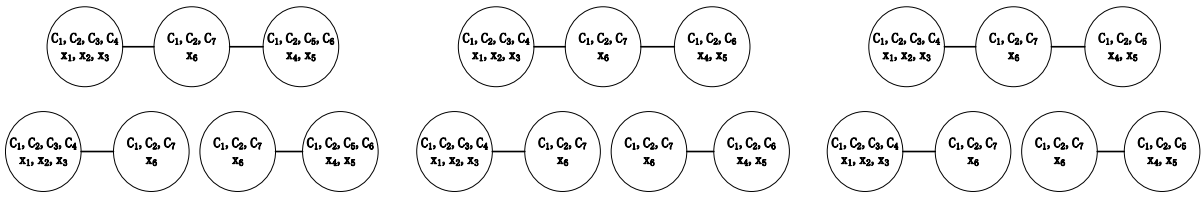
(b) Fixing an assignment to the variables in the middle bag results in two independent instances.

Figure 1: An example showing bounded tree-width **SAT** can be solved efficiently

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). 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_2 (c) ϕ_3

Figure 2: 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 2, 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 $x_4 = 0$, then C_1 can not be satisfied by x_4. Similarly, satisfying C_6 requires $x_5 = 0$, then C_2 can not be satisfied by x_5. In order to find a satisfying truth assignment, when processing the right part, we need the information which of C_1, C_2 is already satisfied in the left part. ϕ_1 is not satisfiable so whichever does not affect the result. 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".

2.4 Notation and terminology

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_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 in the satisfiability problem (and not e.g. in #**SAT**) we observe that d combinations suffice.

3 On the complexity of width-parameterized SAT

We show that **SAT** problems parameterized by path- and tree-width are complete for natural complexity classes. These completeness results, together with a new characterization of the levels of the **NSC** hierarchy yield two important corollaries. First, our Conjecture 1 holds true under a widely believed complexity assumption. Second, under a different well-known complexity assumption ($\mathbf{NL} \subsetneq \mathbf{SAC}^1$), for the same width parameter $w(|\phi|)$ **SAT** of tree-width $O(w(|\phi|))$ cannot be efficiently reduced to **SAT** of path-width $O(w(|\phi|))$.

3.1 More preliminaries and notation

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

Conjecture 2. *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* $\mathbf{NL} \not\subseteq \mathbf{TISP}(n^{O(1)}, n^{o(1)})$. *This implies the weaker conjecture* $\mathbf{SAC}^1 \not\subseteq \mathbf{TISP}(n^{O(1)}, n^{o(1)})$.

We denote by $\mathbf{SAT}_{\mathsf{tw}}(w(|\phi|))$ 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 $\mathbf{SAT}_{\mathsf{pw}}(w(|\phi|))$. [Pap09] shows that $\mathbf{SAT}_{\mathsf{pw}}(w(|\phi|))$ is complete for the class $\mathbf{NL}[\frac{w(|\phi|)}{\log|\phi|}]$, characterized by log-space bounded Turing Machines augmented with a polynomially long read-only, non-deterministic tape on which they make $O(\frac{w(|\phi|)}{\log|\phi|})$ passes.

A semi-unbounded circuit is a circuit with unbounded fan-in OR gates, bounded AND gates and all the negations at the input level. We use the notation $\mathbf{SAC}(\mathsf{depth}, \mathsf{size})$, and we define $\mathbf{SAC}^k := \mathbf{SAC}(\log^k n, n^{O(1)})$ and $\mathbf{SAC}^k_{\mathsf{quasi}} := \mathbf{SAC}(\log n, 2^{O(\log^k n)})$. The study of **SAC** circuits, and the various classes $\mathbf{SAC}^i$ has received considerable attention e.g. [BCD$^+$89, Ven87]. The $\mathbf{SAC}^k_{\mathsf{quasi}}$ classes (very shallow quasi-polynomial size circuits) are introduced in this paper; they characterize the **NSC** hierarchy (Equation (2)). For these families of circuits we use Dlogtime-uniformity [BIS90]. 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 $t(\in \{AND, OR, 0, 1\})$ 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 $\mathbf{SAC}^k_{\mathsf{quasi}}$ 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 ultimately related to space-time bounded non-deterministic Auxiliary Pushdown Automata (NAuxPDAs). A NAuxPDA is a non-deterministic space-bounded Turing Machine equipped with an unbounded stack (see [Coo71] for a precise definition). $\mathbf{NAuxPDA}(s(n), t(n))$ is the class of decision problems decidable by an NAuxPDA in space $O(s(n))$ and time $O(t(n))$. Generalizing the arguments in [Ruz80] and [Ven87] we obtain:

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

Proof sketch. The proof of $\mathbf{SAC}^k_{\mathsf{quasi}} \supseteq \mathbf{NAuxPDA}(\log^k n, n^{O(1)})$ is a series of several simulations, generalizing the characterization theorems in [Ruz80, Ven87], which proved a down-scaled version of the lemma, namely, $\mathbf{NAuxPDA}(\log n, n^{O(1)}) = \mathbf{SAC}(\log n, n^{O(1)})$.

The route of simulations follows exactly the same way as in the previous works, except that more careful analysis is required. The details and some definitions (e.g. alternation, Alternating Turing Machine, treesize bounded alternation) are beyond the scope of the main topic of this paper and since we make no major change in these arguments, they are omitted. Interested readers are referred to [Ruz80, Ven87]. For completeness, we provide an overview of the simulation.

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

The high-level of the simulation witnessing $\mathbf{NAuxPDA}(\log n, n^{O(1)}) \subseteq \mathbf{SAC}(\log n, n^{O(1)})$ is as follows. First, a machine deciding a set in $\mathbf{NAuxPDA}(\log^k n, n^{O(1)})$ is simulated by a semi-unbounded ATM which runs simultaneously in $O(\log^k n)$ space, $O(\log^{k+1} n)$ time, and $O(\log n)$ alternations. This ATM is further simulated by a class of uniform semi-unbounded circuits with size $2^{O(\log^k n)}$, $O(\log^{k+1} n)$ depth and $O(\log n)$ alternations (in fact, this step is rather straightforward). The last simulation shrinks the gates between two alternations to constant depth using semi-unboundedness and the so-called *reachability sub-circuits*. $O(\log n)$ alternations directly result in $O(\log n)$ depth, while the reachability sub-circuits only need to test whether two OR gates are connected by a path of at most $O(\log^{k+1} n)$ OR gates and therefore have depth $O(\log \log n)$. The Dlogtime-uniformity of the circuit is immediate.

We prove the other direction, namely $\mathbf{NAuxPDA}(\log n, n^{O(1)}) \supseteq \mathbf{SAC}(\log n, n^{O(1)})$ by showing that $\mathbf{SAT}_{\mathsf{tw}}(\log^k |\phi|)$ is hard for $\mathbf{SAC}(\log n, n^k)$, and is contained in $\mathbf{NAuxPDA}(\log n, n^{O(1)})$. These are presented later in Lemma 4 and Lemma 5. □

The reader may be surprised that acceptance of a super-polynomial size circuit can be verified in (non-deterministic) 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 5 for details).

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

In Theorem 1 we show that $\mathbf{SAT}_{\mathsf{pw}}(\log^k |\phi|)$ is complete for $\mathbf{NSC}^k$ and Theorem 2 states that $\mathbf{SAT}_{\mathsf{tw}}(\log^k |\phi|)$ is complete for $\mathbf{SAC}^k_{\mathsf{quasi}}$. 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 logspace. Putting these together we have the following characterization of the **NSC** levels:

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

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 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 a machine can determine if the assignments are consistent without using any additional space. 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 1.

Lemma 2. $\mathbf{NSC}^k = \mathbf{NL}[\log^{k-1} n]$, *for* $k \in \mathbb{Z}^+$.

Proof. Let's see why $\mathbf{NSC}^k \subseteq \mathbf{NL}[\log^{k-1} n]$ first. Let M be a machine that accepts a language $L \in \mathbf{NSC}^k$. 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 $\mathbf{NL}[\log^{k-1} n]$ that is contained in $\mathbf{NSC}^k$. $\mathbf{SAT}_{\mathsf{pw}}(\log^k |\phi|)$ is such a problem, by the following characterization:

Lemma 3 ([Pap09])**.** $\mathbf{SAT}_{\mathsf{pw}}(\log^k |\phi|)$ *is complete for* $\mathbf{NL}[\log^{k-1} n]$*, for* $k \in \mathbb{Z}^+$*, under log-space many-to-one reductions.*

A nondeterministic machine M'' for $\mathbf{SAT}_{\mathsf{pw}}(\log^k |\phi|)$ 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 and Lemma 2, immediately yield the following theorem:

Theorem 1. $\mathbf{SAT}_{\mathsf{pw}}(\log^k |\phi|)$ *is complete for* $\mathbf{NSC}^k$*, for* $k \in \mathbb{Z}^+$*, under log-space many-to-one reductions.*

Theorem 2. $\mathbf{SAT}_{\mathsf{tw}}(\log^k |\phi|)$ *is complete for* $\mathbf{SAC}^k_{\mathit{quasi}}$*, for* $k \in \mathbb{Z}^+$*, under log-space many-to-one reductions.*

Proof. Containment is by Lemma 4 and Lemma 1, and hardness is by Lemma 5.

Lemma 4. $\mathbf{SAT}_{\mathsf{tw}}(\log^k |\phi|) \in \mathbf{NAuxPDA}(\log^k n, n^{O(1)})$

Proof. The algorithm witnessing this containment is very natural when expressed as a **NAuxPDA**; it is a modification of the algorithm in [GP08] with an additional trick to handle arbitrary CNF clauses, and has a very similar structure to the proof of $\mathbf{SAT}_{\mathsf{pw}}(\log^k |\phi|)$ is in $\mathbf{NSC}^k$.

The **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 **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 **NAuxPDA** will first check if v has any children. If not, the **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 **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 **NAuxPDA** will pop the information for the second child off of the stack and search it. If both subtrees are successfully searched, then the **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. □

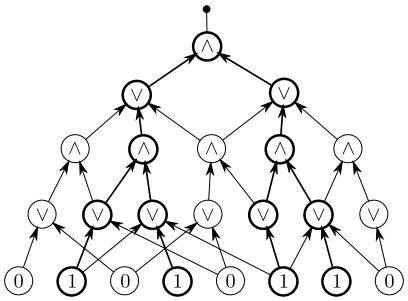
Hardness is more interesting. We do a reduction from an arbitrary language in $\mathbf{SAC}^k_{\mathsf{quasi}}$. Similar "generic reductions" (i.e. reducing the computation of families of **SAC** circuits) for treewidth-related problems have appeared before, e.g. [GLS01] .

Lemma 5. $\mathbf{SAT}_{\mathsf{tw}}(\log^k |\phi|)$ *is hard for* $\mathbf{SAC}^k_{\mathsf{quasi}}$*, under logspace many-to-one reductions.*

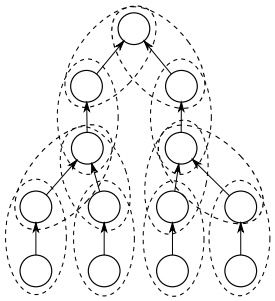
Proof. Fix $L \in \mathbf{SAC}^k_{\mathsf{quasi}}$ and an input x. Let C be the associated **SAC** circuit, with uniformity realized by a Turing Machine M (i.e. the machine that decides the direct connection language). We construct a formula ϕ that is satisfiable if and only if $C(x) = 1$. Without loss of generality we assume the 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 *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 3a 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 3b). 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. 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



(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 3: In 3b a $\mathbf{SAT}_{\mathsf{tw}}(\log^k |\phi|)$ 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.

have: $M(\langle n, x_v, 0, \overline{0}, AND\rangle) = 1$, $M(\langle n, x_v, d_v, x_{u_l}, OR\rangle) = 1$, and $M(\langle n, x_v, d_v, x_{u_r}, OR\rangle) = 1$. When v is at an odd-layer, and u is its child, we have $M(\langle n, x_v, 0, \overline{0}, OR\rangle) = 1$, and either $M(\langle n, x_v, d_v, x_u, AND\rangle) = 1$ 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 inputs bits provided to a Turing machine M' (a modification of M) having running time $O(\log^k n)$ on s. This can be encoded à la Cook-Levin (see e.g. [AB09]) as a CNF of size $O(\log^k n)$. 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 3b, 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 $O(\log^k n)$. □

□

3.3 Evidence for Conjecture 1, and the separation of $\mathbf{SAT}_{\mathsf{pw}}(\log^k |\phi|)$ from $\mathbf{SAT}_{\mathsf{tw}}(\log^k |\phi|)$

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

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

In particular, when $k = 1$, we have that Conjecture 1 for tree-width $O(\log |\phi|)$ is equivalent to $\mathbf{SAC}^1 \not\subseteq \mathbf{TISP}(n^{O(1)}, n^{o(1)})$.

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

[4]Philosophically, the assumption $\mathbf{SAC}^k_{\mathsf{quasi}} \not\subseteq \mathbf{TISP}(2^{O(\log^k n)}, n^{o(1)})$ is not really different than the widely-believed

Corollary 2. *Conjecture 1 for tree-width* $\mathbf{polylog}(|\phi|) \implies \mathbf{SAC}^1 \not\subseteq \mathbf{SC}$.

Thus, modulo these complexity assumptions this settles the lower bound of the Alekhnovich-Razborov question. Note that Corollary 2 opens new avenues for propositional proof complexity [BBI11]; i.e. validating our conjecture for restricted types of algorithms implies progress towards $\mathbf{NC} \neq \mathbf{SC}$.

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

Corollary 3. $\mathbf{SAT}_{\mathsf{tw}}(\log|\phi|)$ *is not log-space reducible to* $\mathbf{SAT}_{\mathsf{pw}}(\log|\phi|)$*, unless* $\mathbf{NL} = \mathbf{SAC}^1$.

In fact, the above holds up to **NL**-reductions. This corollary extends to every poly-logarithmic width under the scaled assumption $\mathbf{NSC}^k \subsetneq \mathbf{SAC}^k_{\mathsf{quasi}}$. 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 [GP08], 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 $\left(2^{2^{\mathcal{TW}(\phi)}}|\phi|^{O(1)}, 2^{\mathcal{TW}(\phi)}|\phi|^{O(1)}\right)$, whereas the space-efficient one works in time-space $\left(3^{\mathcal{TW}(\phi)\log|\phi|}|\phi|^{O(1)}, |\phi|^{O(1)}\right)$. The first one [SS10] 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.

Here is an overview of the time- and space- efficient algorithms.

The time-efficient algorithm does dynamic programming using the tree-decomposition in a typical way [Bod93]: root the tree to make it a binary tree, then for each bag define a $2^{\mathcal{TW}(\phi)}$ size Boolean array, each entry corresponds to the satisfiability of the subformula rooted at the bag with the bag being assigned the assignment indexed by the entry. Clearly, computing the array for the root will solve the satisfiability of the formula, and indeed by the property of a tree decomposition, computing is leaves-to-root fashion we determine the required values.

To simplify this overview of the space-efficient algorithm we shall 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

assumption $\mathbf{SAC}^1 \not\subseteq \mathbf{TISP}(n^{O(1)}, n^{o(1)})$. By analogy let us consider $\mathbf{P} \neq \mathbf{NP}$ and $\mathbf{E} \neq \mathbf{NE}$. It is true that $\mathbf{E} \neq \mathbf{NE}$ is stronger in the sense that $\mathbf{E} \neq \mathbf{NE}$ implies $\mathbf{P} \neq \mathbf{NP}$ (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 [BWMR82]. 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 $\mathbf{SAC}^1 \not\subseteq \mathbf{TISP}(n^{O(1)}, n^{o(1)})$ for logarithmic tree-width, whereas for larger tree-width we have only shown equivalence to the scaled analogs of $\mathbf{SAC}^1 \not\subseteq \mathbf{TISP}(n^{O(1)}, n^{o(1)})$.

[5] This is not a mild assumption, since the space-efficient algorithm does not traverse the tree in some "contiguous manner" (e.g. from leaves to the root). Certain ways of removing the assumption *severely affect* the running time of the algorithm.

this bag. The algorithm works by enumerating and checking recursively 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 4.1 (Lemma 6 below) we show that there always exists a good choice for a bag, reminiscent to the well-known "$\frac{1}{3}$ - $\frac{2}{3}$ lemma" for binary trees. The lemmas in Section 4.1 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 cutting 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 many 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 than the one from the discretization factor ε; i.e. our most general tradeoff algorithm is controlled by two parameters.

4.1 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 4.2) and tradeoff algorithms (Sections 4.3 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 discretized assignment which is crucial in the tradeoff algorithms.

Definition 3 (Splitting operation)**.** *Let $T = (V, E)$ be a tree, and $v \in V$.* Splitting T at v *is the following operation. Let $T_1, \ldots, T_k$ be the trees after removing v from T. The splitting operation results in a forest $\{v\} \cup T_1, \ldots, \{v\} \cup T_k$, 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. The 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, $1 \leq i \leq k$. 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 6. *Consider a tree of size N, a leaf s and $0 < \alpha < 1$. Then, there is a node p where after we split at p, the tree which contains s is of size $\leq \lceil \alpha N \rceil$ and every other tree is of size $\leq \lceil (1-\alpha)N \rceil$. 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 $\langle s \equiv v_1, v_2, \ldots, v_l \rangle$ 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, $1 \leq i \leq l$. It is not hard to see that $a_1 = 1$, $a_l = N$, and a_i strictly increases as i increases. Therefore, there must be a j, such that $a_j \leq \alpha N$ 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 $(1-\alpha)N$. □

Corollary 4. *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 $X^* = \cup_{v_i \in S} X_i$, 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, $\mathcal{T}$ splits into subtrees $\mathcal{T}_i$'s. In every splitting operation, the bag being split is given some assignment. Further suppose $R_{\mathcal{T}}$ is an assignment to $\mathcal{T}$, and $R_{\mathcal{T}_i}$ is an assignment to the subtree $\mathcal{T}_i$. $R_{\mathcal{T}}$ and $R_{\mathcal{T}_i}$'s are said to be *consistent* if

(1) for every i, the bits corresponding to a variable x in $R_{\mathcal{T}_i}$ is the same 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.

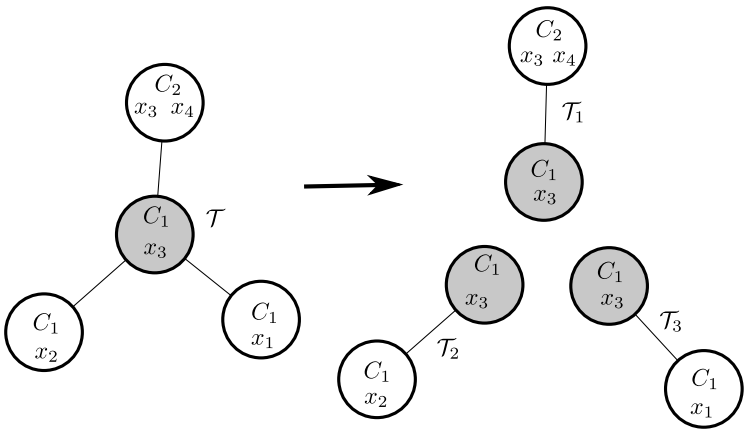


Figure 4: Consistent assignments. $C_1 = x_1 \vee \overline{x_2} \vee x_3$, $C_2 = x_3 \vee \overline{x_4}$. Consider splitting at the gray bag, while fixing the value of the bits, 1 for C_1, 0 for x_3. And possible consistent assignments for $\mathcal{T}_1$, $\mathcal{T}_2$, $\mathcal{T}_3$ would be they all have 0 for x_3, C_1 in $\mathcal{T}_1$ is 1, while in $\mathcal{T}_2$ and $\mathcal{T}_3$ are 0

Remark 2. *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 issue.*

Lemma 7. *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 $d^{\mathcal{TW}(\phi)}$.*

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 3 valid assignments. Recall that 3 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. *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.

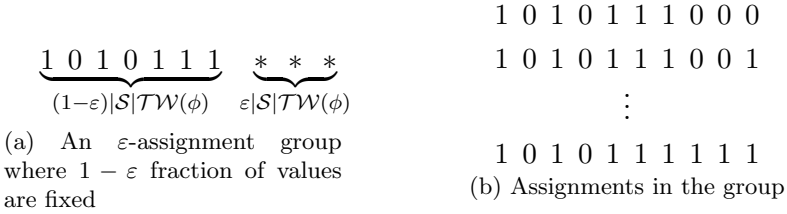
Lemma 8. *An assignment $R_\mathcal{T}$ is* satisfying *if and only if there exist assignments $R_{\mathcal{T}_i}$ to the subtrees $\mathcal{T}_i$, such that the assignments $R_{\mathcal{T}_i}$'s are consistent with $R_\mathcal{T}$ and each of the $R_{\mathcal{T}_i}$ is* satisfying*.*

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, there exists a truth assignment on variables within $\mathcal{T}$. Using the truth assignment, we can always find assignments $R_{\mathcal{T}_i}$'s consistent with $R_{\mathcal{T}}$, such that for these truth assignments the conditions in Definition 4 are met.

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 *satisfiable*. For each subtree $\mathcal{T}_i$, there exists a truth assignment complying to Definition 4. Since all these truth assignments agree on their common variables, we can get a truth assignment from their union, which also meets the axioms in Definition 4. Therefore, the assignment $R_{\mathcal{T}}$ is satisfying. □

An *ε-assignment group* ε-$\mathsf{GR}_{\mathcal{T}}$ is a set of binary strings each of length at most $|S|\mathcal{TW}(\phi)$, in which $1-\varepsilon$ fraction of entries corresponding to the splitting nodes are fixed to some constants. ε-$\mathsf{GR}_{\mathcal{T}}$ and ε-$\mathsf{GR}_{\mathcal{T}_i}$'s are called *consistent* if there exists a way to assign values to each unfixed entry to obtain an assignment for $\mathcal{T}$(as $R_{\mathcal{T}}$) and assignments for $\mathcal{T}_i$'s(as $R_{\mathcal{T}_i}$'s), such that $R_{\mathcal{T}}$ and $R_{\mathcal{T}_i}$'s are consistent. For a tree $\mathcal{T}$ and ε-$\mathsf{GR}_{\mathcal{T}}$, by fixing $(1-\varepsilon)\mathcal{TW}(\phi)$ bits corresponding to variables and clauses contained in the splitting node p, one can derive ε-$\mathsf{GR}_{\mathcal{T}_i}$ for each subtree $\mathcal{T}_i$. Note that the fixed entries for the splitting node p may be different among subtrees, and the unfixed entries in $\mathcal{T}$ need not to be fixed in subtrees. The following important lemma holds, which basically generalizes Lemma 7.



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

(b) Assignments in the group

Figure 5: ε-assignment group

Lemma 9. *The number of distinct ε-$\mathsf{GR}_{\mathcal{T}_i}$'s consistent with ε-$\mathsf{GR}_{\mathcal{T}}$ is at most $d^{(1-\varepsilon)\mathcal{TW}(\phi)}$.*

Proof. 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 three different cases.

Suppose that C does not appear in any previous splitting node. This implies that C only appears in $\mathcal{T}$, then 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.

Suppose that C appears in some previous splitting nodes and its value is fixed in ε-GR_T. If the bit for C is assigned to 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.

Suppose that C appears in some previous splitting nodes, but its value is unfixed. C must appear as unfixed in at least one subtree. Without loss of generality, we assume that C appears in subtrees $\mathcal{T}_1, \mathcal{T}_2 \cdots \mathcal{T}_{e_0}$, where $e_0 \geq 1$. The values of C in $\mathcal{T}_1, \mathcal{T}_2 \cdots \mathcal{T}_{e_0}$ are still unfixed, so there are $d_0 - e_0 + 1 \leq d_0$ possible assignments of C in the subtrees $T_{e_0+1}, \cdots, T_{d_0}$, the first one sets all to 0, and the $i(\geq 2)$-th one sets the bit of C in the subtree $\mathcal{T}_{i+e_0-1}$ to 1 and the rest to 0.

Since there are at most $(1-\varepsilon)\mathcal{TW}(\phi)$ unfixed values in p, the number of different combinations of ε-$GR_{\mathcal{T}_i}$ consistent with ε-$GR_{\mathcal{T}}$ is at most $d^{(1-\varepsilon)\mathcal{TW}(\phi)}$ □

4.2 The space-efficient algorithm

The description of the space-efficient algorithm is in Algorithm 1. $\mathcal{T}$ 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 2. The correctness of the algorithm directly follows by Lemma 8.

Algorithm 1 SAT($\mathcal{T}$, $R_{\mathcal{T}}$)

```
1: if all nodes in T are previous splitting nodes then
2:   if every clause in R_T which assigned to 1 is satisfied by some variables in T then
3:     return true
4:   else
5:     return false
6:   end if
7: else
8:   find the splitting node s according to Corollary 4, and split at s, which results in the subtrees
     T_i
9:   for all assignments R'_{T_i} s consistent with R_T do
10:    if for each subtree T_i, SAT(T_i, R_{T_i}) = true then
11:      return true
12:    end if
13:  end for
14:  return false
15: 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 7

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

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

4.3 Tradeoff Algorithms

We present a family of algorithms for bounded tree-width **SAT** described in Algorithm 2. Different choices for the splitting algorithm (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 ε-$\mathsf{GR}_{\mathcal{T}}$, and returns an array $M(\mathcal{T}, \varepsilon\text{-}\mathsf{GR}_{\mathcal{T}})$ of $2^{\varepsilon|\mathcal{S}|\mathcal{TW}(\phi)}$ entries, where the ith entry indicates whether the i-th assignment of ε-$\mathsf{GR}_{\mathbf{T}}$ can be satisfied.

Algorithm 2 SAT-tradeoff($\mathcal{T}$, $\mathcal{S}$, ε-GR$_{\mathcal{T}}$)

$M(\mathcal{T}, \varepsilon\text{-GR}_{\mathcal{T}}) \leftarrow$ all zero matrix
if all nodes in $\mathcal{T}$ are previous splitting nodes **then**
  for $j \leftarrow 1$ **to** $|\varepsilon\text{-}GP_{\mathcal{T}}|$ **do**
    Let $R_{\mathcal{T}}$ be the jth assignment in ε-GR$_{\mathcal{T}}$
    if all entries for a clause in $R_{\mathcal{T}}$ assigned to 1 are satisfied by some variables in $\mathcal{T}$ **then**
      $M(\mathcal{T}, \varepsilon\text{-GR}_{\mathcal{T}})_j \leftarrow 1$
    end if
  end for
else
  Split at the node returned by a splitting algorithm given $\mathcal{T}$, $\mathcal{S}$
  Denote the subtrees after the splitting as $\mathcal{T}_i$'s
  for all ε-group assignments ε-GR$_{\mathcal{T}_i}$ $\forall i$, which are all consistent with ε-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$(P, \mathcal{T}_i, \varepsilon\text{-GR}_{\mathcal{T}_i})$
    for $j \leftarrow 1$ **to** $|\varepsilon\text{-}GP_{\mathcal{T}}|$ **do**
      Let $R_{\mathcal{T}}$ be the jth assignment in ε-GR$_{\mathcal{T}}$
      for all $R_{\mathcal{T}_i} \in \varepsilon\text{-GR}_{\mathcal{T}_i}$, $\forall i$ **do**
        if $M(\mathcal{T}_i, R_{\mathcal{T}_i}) = 1, \forall i$ **and** $R_{\mathcal{T}_i}$'s are all 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 3 has the property that it never creates *type*$_\ell$ tree, $\forall \ell \geq 3$.

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 3. SAT *of tree-width $\mathcal{TW}(\phi)$ can be solved in simultaneously $O(d^{1.441(1-\varepsilon)\mathcal{TW}(\phi)\log|\phi|}|\phi|^{O(1)})$ time and $O(2^{2\varepsilon\mathcal{TW}(\phi)}|\phi|^{O(1)})$ space, where ε is a free parameter, $0 < \varepsilon < 1$.*

Proof. Denote by $T_1(N)$, $T_2(N)$ the running time of $\mathcal{H}_2$ on *type*$_1$ or *type*$_2$ tree 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

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

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

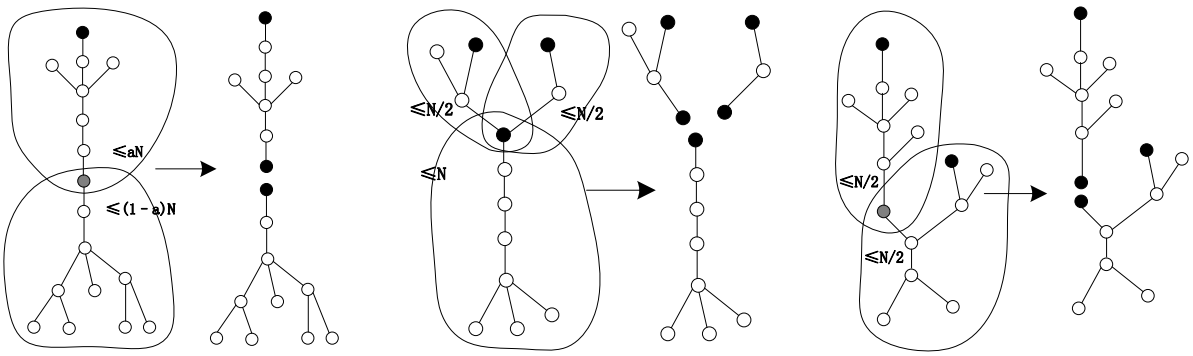


Figure 6: Finding the splitting node in three different cases.

Algorithm 3 Splitting algorithm $\mathcal{H}_2$ with $\mathcal{T}, \mathcal{S}$ as parameters

if $\mathcal{T}$ with $\mathcal{S}$ is a $type_0$ tree **then**
  return the 1/2-splitting node
else if $\mathcal{T}$ with $\mathcal{S}$ is a $type_1$ tree **then**
  consider the previous splitting node as the root
  return the α-splitting node
else if $\mathcal{T}$ with $\mathcal{S}$ is a $type_2$ tree **then**
  suppose the two splitting nodes are p_1 and p_2
  consider p_1 as the root and compute the 1/2-splitting node m
  if m is on the path between p_1 and p_2 **then**
    return m
  else
    return the least common ancestor c of m and p_2
  end if
end if

two $type_2$ trees with size at most $N/2$ and several $type_1$ trees. Hence:

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

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

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

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

4.4 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 $type_i, \forall i \geq 3$ trees. Interestingly, it can be shown that this specific splitting algorithm is optimal over all splitting strategies which enjoy this property.

Definition 5. *Denote by $\mathfrak{A}_c$ ($\forall c \geq 2$) the family of algorithms for* **SAT** *with bounded tree-width following the framework in Algorithm 2 which use a splitting algorithm without creating $type_i$ trees $\forall i > c$.*

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

Definition 6. *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 the root r and the root of subtree F_h.*

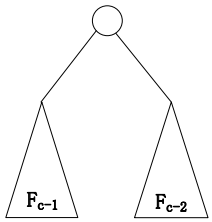


Figure 7: An *h-Fibonacci tree* (F_h).

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, r)-Fibonacci tree with N nodes, where r is the splitting node, $h = \lceil \log_{(1+\sqrt{5})/2} N \rceil$. We prove that this is hard for any algorithms in $\mathfrak{A}_2$, namely,

Theorem 4. *Every algorithm in $\mathfrak{A}_2$ runs in $\Omega(3^{1.441(1-\varepsilon)\mathcal{TW}(\phi)\log N}|\phi|^{\Theta(1)})$ time on the instance constructed above.*

Proof. We define a special *type*$_1$ tree and a special *type*$_2$ tree: $\mathcal{T}_{1,h}$ and $\mathcal{T}_{2,h}$. $\mathcal{T}_{1,h}$ is constructed by a splitting node connected to the root of a subtree F_h, and $\mathcal{T}_{2,h}$ is constructed by two separate splitting nodes connected to another node which roots subtree F_h. We prove by induction that the running time for $\mathcal{T}_{1,h}$ is $\Omega(3^{(1-\varepsilon)\mathcal{TW}(\phi)h}|\phi|^{\Theta(1)})$, and the running time for $\mathcal{T}_{2,h}$ is $\Omega(3^{(1-\varepsilon)\mathcal{TW}(\phi)(h+1)}|\phi|^{\Theta(1)})$. Base cases are vacuous where $h \leq 2$. Suppose the statement is correct for any $h_0 < h$. For $\mathcal{T}_{1,h}$, by induction hypothesis, if we split at the root of F_h, the running time is $\Omega(3^{(1-\varepsilon)\mathcal{TW}(\phi)(1+(h-1))}|\phi|^{\Theta(1)})$, if we split at some node inside the subtrees F_{h-1} or F_{h-2}, the running time is $\Omega(3^{(1-\varepsilon)\mathcal{TW}(\phi)(1+(h-2)+1)}|\phi|^{\Theta(1)})$. So the running time for $\mathcal{T}_{1,h}$ is $\Omega(3^{(1-\varepsilon)\mathcal{TW}(\phi)h}|\phi|^{\Theta(1)})$. For tree $\mathcal{T}_{2,h}$, we must split at the node connecting two splitting nodes, so again by induction hypothesis the running time is $\Omega(3^{(1-\varepsilon)\mathcal{TW}(\phi)(h+1)}|\phi|^{\Theta(1)})$.

Since the number of nodes in F_h is $\Omega((\frac{1+\sqrt{5}}{2})^h)$, by what is shown in the previous paragraph, the running time for the instance constructed above can be lower bounded by $\Omega(3^{\frac{1}{-\log(1-\alpha)}(1-\varepsilon)\mathcal{TW}(\phi)\log N}|\phi|^{\Theta(1)})$, where $\alpha = \frac{3-\sqrt{5}}{2}$ (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 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. *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 ($3^{(\lambda_c(\log N - c)+c)(1-\varepsilon)\mathcal{TW}(\phi)}|\phi|^{O(1)}$, $2^{c\varepsilon\mathcal{TW}(\phi)}|\phi|^{O(1)}$) 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: λ_c for small c's

5.1 Generalized tradeoff algorithms

We have already seen a tradeoff algorithm which avoids $type_c$ trees for $c \geq 3$. It is natural to ask if the algorithm can be generalized to allow up to $type_c$ trees for some $c \geq 3$, 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 $1 \leq i \leq c$, 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 $type_j$ ($j \leq i$) 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 line 10 in Algorithm 2.

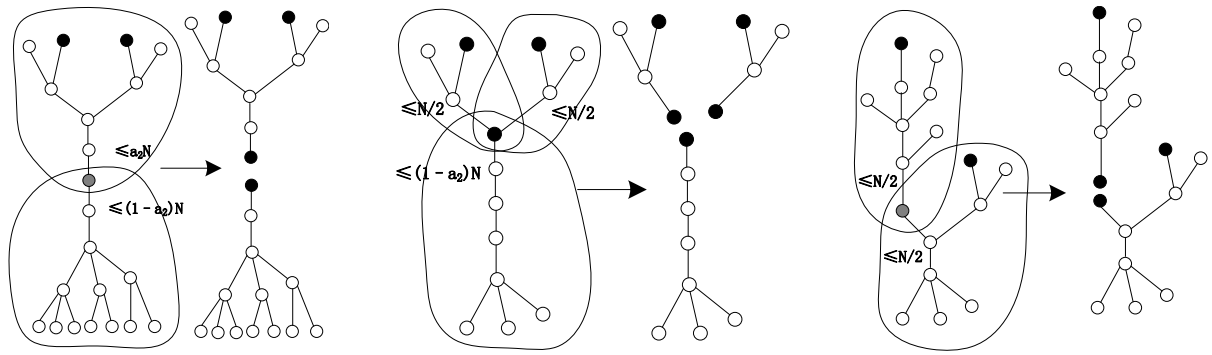


Figure 8: Finding the splitting node in three different cases.

Each $\alpha_{c,i}$ for any $1 \leq i < c$ is a parameter satisfying $0 \leq \alpha_{c,i} \leq 1/2$. To prevent $type_{c+1}$ trees, splitting nodes of $type_c$ trees must be on the path between some pair of existing 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 (see page 23).

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

Definition 7. *c-splitting depth $\mathsf{SD}_c(\mathcal{A}, \mathcal{T}, S)$ of a splitting algorithm $\mathcal{A}$ on tree T with previous splitting nodes S is inductively defined as follows, where the case for $|S| > c$ is for well-definiteness:*

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

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.

c-minimal splitting depth $\mathsf{MSD}_c(\mathcal{T}, S)$ is 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 follows by

Algorithm 4 Splitting algorithm $\mathcal{H}_c$ with $\mathcal{T}, \mathcal{S}$ as arguments

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 the number of nodes in $\mathcal{T}$ is less than 2^{c-i} **then**
    return the 1/2-splitting node
  else
    arbitrarily pick a previous splitting node as root
    compute a $\alpha_{c,i}$-splitting node q_1
    if q_1 is not on the path between any pair of previous splitting nodes **then**
      return q_1
    else
      compute a 1/2-splitting node q_2.
      if q_2 is not on the path between any pair of previous splitting nodes **then**
        return the least common ancestor of q_2 and all previous cutting nodes
      else
        return q_2
      end if
    end if
  end if
end if

two technical lemmas: Lemma 10 establishes the recurrences according to the recursive algorithm, and Lemma 11 deals with choice of parameters. For simplicity of presentation we ignore issues regarding the divisibility of N by 2.

Lemma 10. *For every $c \geq 2$, tree $\mathcal{T}$ with N nodes and splitting nodes S, let $D_{c,|S|}(N) = \mathsf{SD}_c(\mathcal{H}_c, \mathcal{T}, S)$. Then for each $1 \leq i < c$:*

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

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 $1-\alpha_{c,i} > 1/2$, 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:

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

□

Lemma 11. $\mathsf{SD}_c(\mathcal{H}_2, \mathcal{T}, \emptyset)$ *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. Let $D'_{c,i}(N)$ be a function satisfying the following equations,

$$\begin{aligned} D'_{c,i}(N) &= D'_{c,1}((1-\alpha_{c,i})N) + 1 = D'_{c,i+1}(\alpha_{c,i}N) + 1, \text{for } 1 \leq i < c \\ D'_{c,c}(N) &= D'_{c,1}(N) + 1 \end{aligned}$$

By manipulating the first equation, we can derive that for $i : 1 < i \leq c$, $D'_{c,i}(N) = D'_{c,1}((1-\alpha_{c,i-1})N/\alpha_{c,i-1})$. 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,

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

Since $D'_{c,1}(N) = D'_{c,1}((1-\alpha_{c,1})N) + 1$, to minimize the values of $D'_{c,1}$ we let $1 - \alpha_{c,1} = \alpha_{c,i}(1-\alpha_{c,i+1})/(1-\alpha_{c,i})$, and by rearranging $\alpha_{c,i+1} = 1 - (1-\alpha_{c,1})(1-\alpha_{c,i})/\alpha_{c,i}$. Inductively, the following can be proved

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

Now look at the border conditions, since $D'_{c,c}(N) = D'_{c,1}(N) + 1 = D'_{c,1}((1-\alpha_{c,c-1})N/\alpha_{c,c-1})$. Again, to minimize the values of $D'_{c,1}$, let $\alpha_{c,c-1}/(1-\alpha_{c,c}) = 1 - \alpha_{c,1}$, and therefore $\alpha_{c,c-1} = (1-\alpha_{c,1})/(2-\alpha_{c,1})$. Thus, $(1-\alpha_{c,1})/(2-\alpha_{c,1}) = 1 - \frac{\alpha_{c,1}(1-\alpha_{c,1})^{c-1}}{2\alpha_{c,1}-1+(1-\alpha_{c,1})^{c-1}}$, which implies

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

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

$$\begin{aligned} D'_{c,i}(N) &= \max\{D'_{c,1}((1-\alpha_{c,i})N), D'_{c,i+1}(\alpha_{c,i}N), D'_{c,i}(N/2)\} + 1, 1 \leq i < c \\ D'_{c,c}(N) &= \max\{D'_{c,1}(N), D'_{c,c}(N/2)\} + 1 \end{aligned}$$

Combining with Lemma 10, it can be proved by induction that $D'_{c,i}(N)$ upper bounds $D_{c,i}(N)$ for all c, i. And by the choice of the parameters, the recurrence can be solved using standard tools. Specifically,

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

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 $\sum_{l=1}^{c} 2^{-\frac{l}{\lambda_c}} = 1$. □

Proof. (Proof of Theorem 5) 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 $type_i$, $\forall i \leq c$ trees are allowed. □

The value λ_c depending on the choice of parameter c seems quite artificial in the analysis of our algorithms. Here is an upper bound on λ_c.

Lemma 12. $\lambda_c < 1 + \frac{2}{2^{c/2}}$

Proof. Let $f(X) = X^c - \sum_{i=0}^{c} X^i$, 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 $f(2 - \frac{1}{2^{c/2}}) < 0$ then there must be a root between 2 and $2 - \frac{1}{2^{c/2}}$. Denote $y = 2 - \frac{1}{2^{c/2}}$,

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

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

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 (of Theorem 5)**.** *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 fixed ε and c, by Theorem 5, there is an algorithm with running time $O(d^{\lambda_c(1-\varepsilon)\log_2 N\mathcal{TW}(\phi)}|\phi|^{O(1)})$ and space $O(2^{c\varepsilon\mathcal{TW}(\phi)}|\phi|^{O(1)})$ for any $\varepsilon > 0$. Set $\varepsilon = \frac{\varepsilon'}{c}$, then the space is $O(2^{\varepsilon'\mathcal{TW}(\phi)}|\phi|^{O(1)})$ and the running time is $O(d^{\lambda_c(1-\frac{\varepsilon'}{c})\log_2 N\mathcal{TW}(\phi)}|\phi|^{O(1)})$. By Lemma 12, $\lambda_c(1-\frac{\varepsilon'}{c}) < (1+\frac{2}{2^{c/2}})(1-\frac{\varepsilon'}{c}) < 1$ for sufficiently large c. □

5.2 Optimality of the generalized tradeoff algorithm

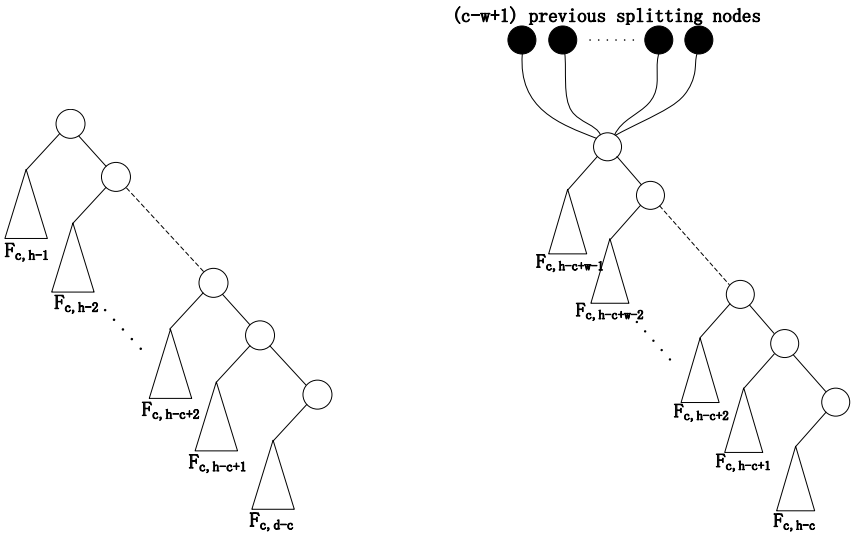
Similarly to the last part of the previous section (Section 4), we also prove the optimality of the generalized tradeoff algorithm. However, in this case the matching lower bound is more surprising (since the upper bound involved a lot of guessing). We construct the hard instance using extended generalized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

Definition 8. *For any integer $c \geq 2$, and a 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 $F_{c,h}$.*



(a) A (c, h)-Fibonacci tree (b) $G_{c,h,w}$

Figure 9: Illustration of the hard instances used in proving optimality

See Figure 9a for an illustration of a (c, h)-Fibonacci tree. A (c, h)-Fibonacci tree is indeed the hardest input of the splitting algorithm. To be more specific, the following lemma holds.

Lemma 13. *For each $h \geq 1$, $\mathsf{MSD}(F^*_{c,h,r}, \{r\}) \geq h$.*

Proof. For any $c \geq 2$, $h > c$ and $1 \leq w \leq c$, $G_{c,h,w}$ is a tree defined as follows: first construct a chain of length w, then connect $c-w+1$ splitting nodes to the first node of the chain, and 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 splitting nodes connected to the first node of the chain. We prove that $\mathsf{MSD}(G_{c,h,w}, S_{c-w+1}) \geq h - c + w$, which implies the inequality that we need. Specifically, $\mathsf{MSD}_c(F^*_{c,h,r}, \{r\}) = \mathsf{MSD}_c(G_{c,h,c}, S_1) \geq h - c + c \geq h$.

The inequality is proved by induction on h. The base case is trivial. Suppose for any $h < h_0$, $\mathsf{MSD}_c(G_{c,h,c}, S_{c-w+1}) \geq h-c+w$. Now we prove $\mathsf{MSD}_c(G_{c,h_0,c}, S_1) \geq h_0-c+w$ by induction on w. When $w = 1$, to prevent $type_i$ tree for $i > c$, we must split at the first node of the chain. Therefore, $\mathsf{MSD}_c(G_{c,h_0,1}, S_c) = 1 + \mathsf{MSD}_c(G_{c,h_0-c,c}, S_1) \geq h_0 - c + 1$. When $w > 1$, if the splitting node is in the subtree $F_{c,h_0-c+w-1}$ connected to the first node of the chain, $\mathsf{MSD}_c(G_{c,h_0,w}, S_{c-w+1}) \geq 1 + \mathsf{MSD}_c(G_{c,h_0,w-1}, S_{c-w+2}) \geq h_0-c+w$, otherwise $\mathsf{MSD}_c(G_{c,h_0,w}, S_{c-w+1}) \geq 1+\mathsf{MSD}_c(G_{c,h_0-c+w-1,c}, S_1) = h_0 - c + w$. □

Theorem 6. *For every $c \geq 2$ and $N > 2^c$, there exists a tree $\mathcal{T}$ with N nodes, such that $\mathsf{MSD}_c(\mathcal{T}, \emptyset) \geq \lambda_c(\log N - c) + c - O(1)$.*

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 at most constant times of 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 $N > 2^c$, there exists a tree $\mathcal{T}$ with N nodes, such that the c-minimal splitting depth of $\mathcal{T}$ $\mathsf{MSD}_c(\mathcal{T}, \emptyset)$ is at least $\lambda_c(\log N - c) + c - O(1)$. □

Similarly to Theorem 4, here we conclude the optimality of our tradeoff algorithm. That is, for fixed $c \geq 2$, ε, $0 < \varepsilon < 1$, any algorithm in $\mathfrak{A}_c$ there is an instance ϕ, for which the running time is $\Omega(3^{\lambda_c(\log|\phi|-c)+c-O(1)}|\phi|^{\Theta(1)})$.

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 [BBI11] took the first step towards this direction. Such results can be also understood as partial progress towards $\mathbf{SC} \neq \mathbf{NC}$.

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 Kevin Matulef and Alexander Razborov for useful remarks and suggestions. The first author acknowledges the support of NSF Grants CCF-0832787 and CCF-1064785.

References

[AAD+00] M. Agrawal, E. Allender, S. Datta, H. Vollmer, and K. W. Wagner. Characterizing small depth and small space classes by operators of higher type. *Chicago J. Theor. Comput. Sci*, 2000(2), 2000.

[AB09] S. Arora and B. Barak. *Computational complexity: a modern approach*, volume 1. Cambridge University Press, 2009.

[AR02] M. Alekhnovich and A.A. Razborov. Satisfiability, branch-width and Tseitin tautologies. In *Foundations of Computer Science (FOCS)*, pages 593–603. IEEE, 2002.

[BBI11] P. Beame, C. Beck, and R. Impagliazzo. Time-space tradeoffs in resolution: Superpolynomial lower bounds for superlinear space. In *Symposium on Theory of Computing (STOC)*, 2011.

[BCD+89] A. Borodin, S. A. Cook, P. Dymond, L. Ruzzo, and M. Tompa. Two applications of inductive counting for complementation problems. *SIAM Journal on Computing (SICOMP)*, 18(3):559–578, 1989.

[BDP09] F. Bacchus, S. Dalmao, and T. Pitassi. Solving #SAT and Bayesian inference with backtracking search. *J. Artif. Intell. Res. (JAIR)*, 34:391–442, 2009. (also FOCS'03).

[BFK+12] H. L. Bodlaender, F. V. Fomin, A. M. C. A. Koster, D. Kratsch, and D. M. Thilikos. A note on exact algorithms for vertex ordering problems on graphs. *Theory of Computing Systems*, 50(3):420–432, 2012.

[BIS90] D. Mix Barrington, N. Immerman, and H. Straubing. On uniformity within NC^1. *Journal of Computer and System Sciences*, 41(3):274–306, December 1990.

[BL03] E. Broering and S. V. Lokam. Width-based algorithms for SAT and CIRCUIT-SAT: (extended abstract). In *SAT*, volume 2919, pages 162–171, 2003.

[Bod93] H. L. Bodlaender. A tourist guide through treewidth. *Acta Cybern.*, 11(1-2):1–22, 1993.

[Bod98] H.L. Bodlaender. A partial k-arboretum of graphs with bounded treewidth. *Theoretical Computer Science*, 209(1-2):1–45, 1998.

[BWMR82] R. V. Book, C. B. Wilson, and X. Mei-Rui. Relativizing time, space, and time-space. *SIAM J. Comput.*, 11(3):571–581, 1982.

[Coo71] S.A. Cook. Characterizations of pushdown machines in terms of time-bounded computers. *Journal of the ACM (JACM)*, 18(1):4–18, 1971.

[Coo85] Stephen Cook. A taxonomy of problems with fast parallel algorithms. *Information and Control*, 64:2–22, 1985.

[CR79] S. A. Cook and R. A. Reckhow. The relative efficiency of propositional proof systems. *J. of Symbolic Logic*, 44(1):36–50, 1979.

[FK10] F. Fomin and D. Kratsch. *Exact Exponential Algorithms*. Springer, 2010.

[FMR08] E. Fischer, J. A. Makowsky, and E. V. Ravve. Counting truth assignments of formulas of bounded tree-width or clique-width. *Discrete Applied Mathematics*, 156(4):511–529, 2008.

[GLS01] G. Gottlob, N. Leone, and F. Scarcello. The complexity of acyclic conjunctive queries. *Journal of the ACM (JACM)*, 48(3):431–498, 2001.

[GP08] K. Georgiou and P. A. Papakonstantinou. Complexity and algorithms for well-structured k-SAT instances. In *Theory and Applications of Satisfiability Testing - SAT*, pages 105–118, 2008.

[Gro07] M. Grohe. The complexity of homomorphism and constraint satisfaction problems seen from the other side. *Journal of the ACM (JACM)*, 54(1):1–24, 2007. (also FOCS'03).

[Klo94] T. Kloks. *Treewidth: computations and approximations*, volume 842. Springer, 1994.

[KP10] M. Koivisto and P. Parviainen. A space-time tradeoff for permutation problems. In *Symposium on Discrete Algorithms (SODA)*, SODA '10, pages 484–492. SIAM, 2010.

[Mar10] D. Marx. Can you beat treewidth? *Theory Of Computing*, 6:85–112, 2010. (also FOCS'07).

[MS11] R. A. Moser and D. Scheder. A full derandomization of Schöning's k-SAT algorithm. In *Symposium on Theory of Computing (STOC)*, pages 245–252, 2011.

[Pap09] P.A. Papakonstantinou. A note on width-parameterized sat: An exact machine-model characterization. *Information Processing Letters (IPL)*, 110(1):8–12, 2009.

[PPSZ98] R. Paturi, P. Pudlák, M. Saks, and F. Zane. An improved exponential-time algorithm for k-SAT. In *Foundations of Computer Science (FOCS)*, pages 628–637. IEEE, 1998.

[RS83] N. Robertson and P.D. Seymour. Graph minors. I. excluding a forest. *Journal of Combinatorial Theory, Series B*, 35(1):39–61, 1983.

[RS86] N. Robertson and P.D. Seymour. Graph minors. II. algorithmic aspects of tree-width. *Journal of algorithms*, 7(3):309–322, 1986.

[Ruz80] W.L. Ruzzo. Tree-size bounded alternation. *Journal of Computer and System Sciences (JCSS)*, 21(2):218–235, 1980.

[Sch99] T. Schöning. A probabilistic algorithm for k-SAT and constraint satisfaction problems. In *Foundations of Computer Science (FOCS)*, pages 410–414. IEEE, 1999.

[SS10] M. Samer and S. Szeider. Algorithms for propositional model counting. *J. Discrete Algorithms*, 8(1):50–64, 2010.

[Sze04] S. Szeider. On fixed-parameter tractable parameterizations of SAT. In *Theory and Applications of Satisfiability Testing - SAT*, pages 188–202. Springer, 2004.

[Ven87] H. Venkateswaran. Properties that characterize LOGCFL. In *Symposium on Theory of Computing (STOC)*, pages 141–150. ACM, 1987.

[Woe03] G. Woeginger. Exact algorithms for NP-hard problems: A survey. *Combinatorial Optimization—Eureka, You Shrink!*, pages 185–207, 2003.

ECCC ISSN 1433-8092

http://eccc.hpi-web.de