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

August 9, 2012

Abstract

Alekhnovich and Razborov [AR02] presented an algorithm that solves SAT on instances $\phi$ of size $n$ and tree-width $TW(\phi)$, using time and space bounded by $2^{O(TW(\phi))} n^{O(1)}$. Although several follow-ups appeared over the last decade (e.g. [BDP09, BL03, FMR08, SS10]), the first open question of [AR02] remained essentially unresolved: Can one check satisfiability of formulas with small tree-width in polynomial space and time as above? We resolve this question, modulo complexity assumptions (the answer is negative), and we discover a threshold phenomenon.

First, we give a simple algorithm that runs in polynomial space and achieves run-time $3^{TW(\phi)} \log n n^{O(1)}$, nearly matching the run-time of [AR02], but the log $n$ factor in the exponent. Then, we conjecture that this annoying log $n$ factor is in general unavoidable.

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

Although we cannot hope for an improvement asymptotically in the exponent of time and space, we introduce a new algorithmic technique which trades constants in the exponents: for each $\varepsilon$ with $0 < \varepsilon < 1$, we give an algorithm in time-space $(3^{1.441(1-\varepsilon)TW(\phi) \log |\phi| |\phi|^{O(1)}}, 2^{\varepsilon TW(\phi)} |\phi|^{O(1)})$. We systematically study the limitations of our technique for trading off time and space, and we show that our bounds are the best achievable using this technique.

1 Introduction

SAT is the prototypical $NP$-complete problem. In the real-world SAT instances tend to have structure. Also, in practice SAT-solvers abort due to lack of memory. In this paper we provide conclusive, asymptotically tight answers regarding time-space tradeoffs for SAT instances that are structured, where this structure is quantified by tree-width.
This restriction of SAT was studied by Alekhnovitch and Razborov [AR02] (for references prior to this see within), who gave algorithms that work in time $2^{O(TW(\phi))}|\phi|^{O(1)}$ and in space $2^{O(TW(\phi))}|\phi|^{O(1)}$, where $TW(\phi)$ is the tree-width of a CNF formula $\phi$, and $|\phi| = n + m$ where $n$ and $m$ is the number variables and clauses. 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. Modulo complexity assumptions we fully answer this question.

We devise a simple space-efficient algorithm for SAT instances in CNF, which runs in time $3^{TW(\phi)} \log |\phi| |\phi|^{O(1)}$ and space $|\phi|^{O(1)}$. This is the first algorithm with running time exponential in the tree-width of the incidence graph (unlike [BDP09] which is just on the primal graph) of arbitrary CNF instances (unlike [GP08] which is just for $k$-CNFs), that runs in polynomial space. Compared to the question of [AR02] this algorithm suffers a $\log |\phi|$ factor in the exponent of the running time. Our work revolves around this logarithmic factor. First, we conjecture that it cannot be removed:

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

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

Assuming for now that Conjecture 1 holds, it makes sense to devise algorithms that approach these limits. This is the topic of Section 4 which constitutes the more technically involved part of this work, though the practical significance of these algorithms is debatable.

### 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.

**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 \( 2^{O(TW(\phi) \log |\phi|)} \) and space \( |\phi|^{O(1)} \), when \( TW(\phi) = \Omega(\log |\phi|) \). The deficiencies of [GP08] (which we overcome in our current paper) are not only that their algorithm works only for \( k \)-SAT, but also that the constant in the exponent of the running time cannot be bounded in any easy way due to the non-explicitness of the argument presented there. [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 \( (2^{TW(\phi)}|\phi|^{O(1)}, 2^{TW(\phi)}|\phi|^{O(1)}) \). Also, we remark that algorithms (e.g. for graph problems) which replace the tree-width parameter \( TW \) in the exponent by a \( TW^2 \) and at the same time reducing the space to polynomial (see e.g. [LMS11]) are strictly worse than our algorithms (and in particular fail to reach our target lower bound for the Alekhnovitch-Razborov question). In particular, unlike the classical parameterized complexity approach, the interesting part of our treatment (and in particular our complexity results) are for values of tree-width that are related to the input length, and in fact \( \Omega(\log n) \).

**Improving constants, previous work, and what’s different.** Improving the constant in the base of an exponential time algorithm is a well-established goal in the field of exact computation for NP-hard problems; see e.g. [FK10, Woe03] for an overview. In particular, for \( k \)-SAT there is a line of work in algorithms that run in time \( \alpha^n \) 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.

Finally, to the best of our knowledge our work is the first to address the issue of smooth time-space tradeoffs. Prior to our work there are others which solely discuss lower bounds on the running time; e.g. for graph problems (and under the very strong ETH assumption [IPZ01]) [LMS11a]. Another kind of tradeoff (between size of the separator and the sharpness) for graph problems was discussed in [FLGS10]. We should also mention that in two excellent works on constraint satisfaction problems Grohe [Gro07] (assuming that \( \text{FPT} \neq \text{W}[1] \)) and Marx [Mar10] (assuming ETH) essentially show that the running time of the known width-based algorithms is optimal.

**Hardness results and complexity characterizations.** Every problem in \( \text{NP} \) can be seen as a problem where accepting instances can be verified in logarithmic space (i.e. we can settle for less than polynomial time in the verification). \( \text{SAT} \) of bounded path-width has been shown [Pap09] complete for the class of problems that can be decided by a logarithmic space machine which has “streaming access” to the tape containing the witness; i.e. scanning the witness tape at most a given number of times. In particular, \( O(r(|\phi|)) \) passes correspond to \( \text{SAT} \) instances with given path-decompositions of width \( r(|\phi|) \log |\phi| \). Specifically, deciding formulas with given path decompositions of width \( O(\log |\phi|) \) is complete for \( \text{NL} \), and [Pap09] asks whether the complexity of \( \text{SAT} \) instances when the parameter is tree-width \( O(\log |\phi|) \) is more difficult. In this paper we answer this question affirmatively, unless\(^1\) \( \text{NL} = \text{SAC}^1 \). Furthermore, we show an exact correspondence of these “streaming verification” classes with the levels of the known NSC hierarchy. Our new characterization through semi-unbounded circuits is of independent interest, and seems more natural than the characterizations presented in [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 \([\text{CR79}]\) 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 implies that there exists a family of formulas \( \phi \) of tree-width \( TW(\phi) \) where for every \( k > 0 \) every resolution refutation of size \( n^k \) requires space \( 2^{TW(\phi) \frac{\log \log n}{\log \log \log n}} \). This very significant development is the first super-polynomial lower bound of this sort, and through our work it can be interpreted as validating the \( \text{SAC}^1 \subseteq \text{SC} \) conjecture, at least for a class of restricted algorithms. This is a new direction; lower bounds in proof complexity are clearly connected to the \( \text{NP} \neq \text{coNP} \) conjecture [CR79], but have not previously seemed to have a bearing on the \( \text{SC} \neq \text{NC} \) question.

### 1.2 Our contribution and techniques

Semi-unbounded combinatorial circuits play an important role in this work. A semi-unbounded circuit (\( \text{SAC} \)) has AND gates of bounded fan-in, OR gates of unbounded fan-in, and all the negations at the input level. Despite their exotic nature these circuits have essential differences from bounded and unbounded fan-in circuits; see Section 3.1 for a discussion. Although our developments regarding \( \text{SAC} \) circuits are of independent interest, we stress that the \( \text{SAC} \) circuits do not appear

\(^1\) It is conjectured, e.g. [Coo85], that \( \text{SAC}^1 \neq \text{NL} \). Note that \( \text{SAC}^1 \) is also known as \( \text{LOGCFL} \)
in the statements of the lower bounds; i.e. these circuits can be thought of as a technical concept which appears inside our proofs.

In Section 3, we show that SAT of a given tree decomposition of width \( \log^k n \) is complete for the class \( \text{SAC}^k_{\text{quasi}} := \text{SAC}(O(\log n), 2^{O(\log^k n)}) \), 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 \( \text{NSC}^k = \text{NTISP}(n^{O(1)}, O(\log^k n)) \) is contained in \( \text{SAC}^k_{\text{quasi}} \) which is in turn contained in \( \text{NSC}^{k+1} \), where \( \text{NTISP} \) denotes the set of problems decidable by non-deterministic Turing Machines that are simultaneously TIme-SPace bounded. Note that the \( \text{NSC} \) levels are direct space-scaled analogs of NL and these \( \text{SAC} \) classes are direct size-scaled analogs of \( \text{SAC}^1 \). 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 \( \text{LOGCFL} = \text{SAC}^1 \). More importantly, putting these developments together we conclude that our conjecture implies \( \text{NC} \neq \text{SC} \), and in fact as the tree-width ranges over different functions of the input length our conjecture is shown to be equivalent to a resource-scaled analog \( \text{NC} \neq \text{SC} \). Somewhat less rigorously:

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

On the positive side, we give a recursive algorithm that runs in time-space \( (3^{TW(\phi)} \log |\phi|, |\phi|^{O(1)}, |\phi|^{O(1)}) \) (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 \( 0 < r < 1 \) that controls the discretization of the truth assignment space. This family of algorithms is presented in Section 4 and in its full generality in Section 5. In the same sections we show that all of the infinitely-many pairs of values are of interest, depending on the different time-space bounds one may want to achieve.

Remark 1. 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.
2.1 Notation

All logarithms are of base 2, and all propositional formulas are in Conjunctive Normal Form (CNF). SAT is the decision problem where given an arbitrary CNF formula we want to decide if it is satisfiable. We let $k$-SAT denote the restriction of SAT to CNFs where each clause has at most $k$ literals. For a formula $\phi$, $m$ 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. $\bigcup_{i=1}^{|W|} X_i = V$
2. $\forall (i, j) \in E, \exists t \in W$, s.t. $i, j \in X_t$.
3. $\forall v$, the set $\{t : v \in X_t\}$ 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 $\text{TW}(G)$ 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 a path decomposition, and the specific tree-width is called the path-width $\text{PW}(G)$. The following inequality holds (e.g. [Bod98])

$$\text{TW}(G) \leq \text{PW}(G) \leq O(\log |V| \cdot \text{TW}(G)) \quad (1)$$

**Definition 2.** The incidence graph $G_\phi$ of a SAT instance $\phi$ is a bipartite graph, where in one side of the bipartization each node is associated with a distinct variable, and in the other each node is associated with a clause. There is an edge between a clause-node and a variable-node if and only if the variable appears in a literal of the clause. The tree-width of a formula $\phi$ is the tree-width of its incidence graph, $\text{TW}(\phi) = \text{TW}(G_\phi)$. When it is clear from the context we may abuse notation and write $\text{TW}(\phi)$ to denote the width of a given decomposition of $G_\phi$.

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

1. $|W| = O(\text{TW}(\phi) \cdot |V|) = O(\text{TW}(\phi)|\phi|)$
2. The tree $T = (W, F)$ has bounded degree 3.

We call a tree decomposition nice if it satisfies the two properties above. A tree decomposition can be converted to a nice one in linear time (see e.g. [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 2.** We present our results with the parameter $d$. One may replace $d$ by 2 or 3 when the structure of the input decomposition is known to be a path or a (nice) tree.
2.3 Assignments

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

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

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

3 Lower bounds: Unobtainable time-space tradeoffs

We show that (i) our Conjecture is equivalent to a widely believed complexity assumption (and its scaled analogs), and (ii) under a different well-known complexity assumption ($\text{NL} \subsetneq \text{LOGCFL}$), for the same width parameter $w(|\phi|)$ SAT of tree-width $O(w(|\phi|))$ cannot be efficiently reduced to SAT of path-width $O(w(|\phi|))$. Both of these results follow by first proving that SAT parameterized by path- and tree-width is complete for natural complexity classes. To obtain these results, we heavily rely on properties of semi-unbounded combinatorial circuits. In Section 3.1 we give a primer with basic intuition about these circuits. Also, based on properties of these circuits and on the relation between path-width and tree-width in Equation (2) in Section 3.3, we provide a new characterization of $\text{NSC}$.

3.1 A primer on semi-unbounded fan-in circuits

The statements of the lower bounds do not explicitly refer to semi-unbounded families of circuits, but we use them inside the arguments. A circuit is semi-unbounded when it has unbounded fan-in OR gates, bounded fan-in AND gates, and all the negations are at the input level. The class of problems that can be decided by such circuits of depth $O(\log^d n)$ and polynomial size is denoted by $\text{SAC}^i$. Clearly, $\text{NC}^i \subseteq \text{SAC}^i \subseteq \text{AC}^i$, where $\text{NC}$ and $\text{AC}$ denote the complexity classes characterized by the same parameters with bounded fan-in and unbounded fan-in families of circuits, respectively. There is also the issue of uniformity (the reader will be formally reminded about it in the next sub-section). For the moment let us assume that all the circuit families in question are uniform, under a reasonable form of uniformity.

What is special in SAC circuits? Suppose that you are given oracle access to an unbounded circuit $C$ (with all negation gates on the input level) and an input $x$, and you want to verify
recursively that $C(x) = 1$. The standard algorithm would be to start at the output gate and execute the following recursive algorithm: if the current gate $g$ is an OR gate, nondeterministically pick a gate $h$ that feeds into $g$ and verify that $h$ evaluates to 1; if the current gate $g$ is an AND gate with $h_1, \ldots, h_m$ feeding into it, recursively verify that each of the $h_i$ evaluates to 1. An accepting run of this algorithm corresponds to a (possibly huge) tree, called a "proof tree" [Ven87]; if the circuit $C$ has bounded fan-in (or even semi-unbounded fan-in), then the size of this tree is bounded by $2^{\text{depth}}$. In the bounded fan-in case, restricting the depth of $C$ automatically implies restricting the size of $C$; in the semi-unbounded fan-in case this is not true. The important aspect of semi-unbounded fan-in circuits that we will utilize, is that the proof tree can be much smaller than the circuit (unlike the bounded-fan-in case), and has size exponential in the depth (unlike the unbounded-fan-in case).

An observation on uniformity. Consider a family of circuits of size $2^{O(n)}$ and depth $O(\log n)$ which is polynomial time uniform. (Note that $n$ bits are required, merely to write down the name of one of the gates.) Then, for an input $x$, $|x| = n$, the problem of evaluating the membership of the family $C_n(x)$ is in NP. That is, given $x$ we guess a proof as described above and then verify that it is a valid proof. Clearly, every problem $L \in \text{NP}$ can be computed by such a circuit since in $\log n$ depth we can compute every minterm of the characteristic function of $L \cap \{0, 1\}^n$ (i.e. we trivially encode the entire truth table, adding a big OR on top, as a $O(\log n)$ depth semi-unbounded circuit). In other words, NP is precisely the class of problems computed by such exponential size circuits. Observe that if we do not insist on the uniformity, then an arbitrary function can be computed by a $2^{O(n)}$-size and $O(\log n)$-depth semi-unbounded (non-uniform) circuit.

Relations of SAC circuits to other models of computation. Semi-unbounded fan-in circuits are intimately related to Alternating Turing Machines (ATMs) and to Nondeterministic Auxiliary PushDown Automata (NAuxPDAs). We provide definitions later on; for a more detailed treatment see e.g. [Ruz80]. For the moment let us say that an ATM and a NAuxPDA are basically the same thing. Also, recall that an ATM is a nondeterministic Turing Machine with two kinds of nondeterministic states: existential and universal. An existential state is accepting if and only if at least one successor configuration is accepting, whereas a universal state is accepting if and only if each successor configuration is accepting. Although it requires a bit of work to show equivalence [Ven87] it should come at no surprise that proofs for SAC circuits are related to ATM computations.

3.2 Complexity theory notation and some preliminaries

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

Conjecture 2. The NL-complete graph reachability problem\footnote{Given a directed graph $G = (V, E)$ and two designated vertices $s, t \in V$, is $t$ reachable from $s$?} 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 $\text{NL} \not\subseteq \text{TISP}(n^{O(1)}, n^{o(1)})$. This implies the weaker conjecture $\text{SAC}^1 \not\subseteq \text{TISP}(n^{O(1)}, n^{o(1)})$. 

2Given a directed graph $G = (V, E)$ and two designated vertices $s, t \in V$, is $t$ reachable from $s$?
We denote by $\text{SAT}_{pw}(w(|\phi|))$ the problem of deciding $\text{SAT}$ of a given CNF formula together with a tree decomposition of width $w(|\phi|)$. Similarly, for path-width we use the notation $\text{SAT}_{pw}(w(|\phi|))$. Papadimitriou [Pap09] shows that $\text{SAT}_{pw}(w(|\phi|))$ is complete for the class $\text{NL}[\log(|\phi|)]$, characterized by log-space bounded Turing Machines augmented with a polynomially long read-only, nondeterministic tape on which they make $O(\log(|\phi|))$ passes.

We use the notation $\text{SAC}(\text{depth}, \text{size})$, and we define $\text{SAC}^k := \text{SAC}(\log^k n, n^{O(1)})$ and $\text{SAC}_{\text{quasi}}^k := \text{SAC}(\log n, 2^{O(\log^k n)}).$ The study of $\text{SAC}$ circuits, and the various classes $\text{SAC}^i$ has received considerable attention e.g. [BCD+89, Ven87]. The $\text{SAC}_{\text{quasi}}^k$ classes (very shallow quasi-polynomial size circuits) are introduced in this paper; they characterize the $\text{NSC}$ 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 $d$th input of the gate $i$ in the circuit for inputs of length $n$ is of type $t \in \{\text{AND}, \text{OR}, 0, 1\}$ 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 $\text{SAC}_{\text{quasi}}^k$ circuits, questions about connectivity in the circuits for length $n$ can be answered in time $O(\log^k n)$.

Simultaneously depth-size bounded semi-unbounded circuits are intimately related to space-time bounded $\text{NAuxPDA}s$. A $\text{NAuxPDA}$ is a nondeterministic space-bounded Turing Machine equipped with an unbounded stack (see [Coo71] for a precise definition). $\text{NAuxPDA}(s(n), t(n))$ is the class of languages decidable by a $\text{NAuxPDA}$ in space $O(s(n))$ and time $O(t(n))$. Generalizing the arguments in [Ruz80] and [Ven87] we obtain:

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

**Proof.** The proof of $\text{SAC}_{\text{quasi}}^k \supseteq \text{NAuxPDA}(\log^k n, n^{O(1)})$ can be shown by following the proof for the special case $k = 1$ (i.e., $\text{NAuxPDA}(\log n, n^{O(1)}) = \text{SAC}(\log n, n^{O(1)})$) [Ruz80, Ven87]. (See also [Vol99].) However, in the proof of Lemma 5 we will need to assert that the uniform $\text{SAC}_{\text{quasi}}^k$ have certain properties, and thus we follow a different outline here, to establish that those properties hold.

Ruzzo [Ruz80, Theorem 1 & Corollary 7] showed that any language in $\text{NAuxPDA}(\log^k n, n^{O(1)})$ is accepted by a $\text{NAuxPDA}$ respecting these same resource bounds, where additionally the height of the pushdown is $O(\log^{k+1} n)$ (and pushes and pops consist of moving strings of length $O(\log^k n)$ to and from the stack – hence it is useful to think of the stack as having height $\log n$, over “symbols” of length $O(\log^k n)$). It is easy to see that such a machine can also be assumed to be somewhat “oblivious”, in the sense that the positions of the worktape and input heads at time $t$ are the same for all inputs of length $n$. Rossmanith and Niedermeier subsequently improved on this, to show that the $\text{NAuxPDA}$ can be assumed to be completely oblivious, in the sense that the sequences of pushes and pops are also the same for all inputs of length $n$ [NR95, Theorem 28]. (Rossmanith and Niedermeier state their theorems in terms of machines with a logarithmic worktape bound, but their proof works also for larger space bounds, as long as the time is polynomial.) In particular, the pushes and pops follow a very regular pattern, so that the computation is divided into phases corresponding to the height of the stack. The computation starts and ends with stack height zero, and precisely half-way through the computation, the stack height is also zero. Call these three configurations $C_0, D_0$, and $E_0$. The computation from $C_0$ to $D_0$ and from $D_0$ to $E_0$ all takes place with a stack height of at least 1 “symbol” (where the stack “symbols” are of $O(\log^k n)$ bits each);
these are the two “phases” with height 1. In general, there are $2^i$ phases with height $i$, for each $i \leq i_{\text{max}} = O(\log n)$. Each such phase (for $i < i_{\text{max}}$) has some start configuration $C_i$ and end configuration $E_i$ that take place at times that depend only on the input length $n$, and there is a configuration $D_i$ that also has stack height $i$, such that the computations between $C_i$ and $D_i$ and between $D_i$ and $E_i$ have exactly the same length and are both phases with stack height $i+1$. (The phases at height $i_{\text{max}}$ start in a configuration $C$ that has a number $j \leq n$ recorded in it, and ends in a configuration that records the $j$-th input symbol; no stack manipulation occurs in such a phase.)

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

- There is a move from $A$ to $C_{i+1}$ pushing $\gamma$.
- There is a move from $E_{i+1}$ to $B$ popping $\gamma$.

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

It should be clear that the circuit directly simulates the $\text{NAuxPDA}$. For more details about the uniformity of the circuits, we refer the reader to [NR95, Ven87, Vol99].

We prove the other direction, namely $\text{NAuxPDA}(\log^k n, n^{O(1)}) \supseteq \text{SAC}(\log n, 2^{O(\log^k n)})$ by showing that $\text{SAT}_{\text{tw}}(\log^k |\phi|)$ is hard for $\text{SAC}(\log n, 2^{\log^k n})$, and is contained in $\text{NAuxPDA}(\log^k n, n^{O(1)})$. 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 (nondeterministic) polynomial time. This is related to the structure and size of proofs of accepting inputs for semi-unbounded circuits. In particular, the size of such a proof/certificate is exponential in the depth of the circuit (see the proof of Lemma 5 for details).

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

In Theorem 1 we show that $\text{SAT}_{\text{tw}}(\log^k |\phi|)$ is complete for $\text{NSC}^k$ and Theorem 2 states that $\text{SAT}_{\text{tw}}(\log^k |\phi|)$ is complete for $\text{SAC}_k^{\text{ quasi}}$. We remark that the tree-width/path-width relation $\text{PW}(G) \leq \text{TW}(G) \log n$ can be shown via a reduction computable in logspace [BGHK95]. Putting these together we have the following characterization of the NSC levels:

$$
\begin{align*}
\text{NL} & \subseteq \text{SAC}^1 \subseteq \text{NSC}^2 \subseteq \text{SAC}_k^{\text{ quasi}} \subseteq \text{NSC}^3 \subseteq \cdots \subseteq \text{NSC} = \text{SAC}_k^{\text{ quasi}} \\
\text{NSC}^1 & \subseteq \text{SAC}_k^{\text{ quasi}} 
\end{align*}
$$

(2)
Our completeness results require us to present upper bounds on the complexity of \( \text{SAT} \) with small tree-width and path-width. For these upper bounds, we need the notion of \( \text{consistency} \). Since we have extended the notion of assignment to also include assignments to \( \text{clauses} \), we also need to have a correspondingly extended notation of consistency of assignments. The rigorous definition of consistency is deferred until 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 \( \text{clauses} \) are satisfied such that a satisfying truth assignment can be deduced. For this section, it suffices to know that, if assignments for two bags are written on the worktape, then it is very easy to determine if the assignments are consistent. Also, by the connectivity properties of tree decompositions, it suffices to check consistency of neighboring bags.

Now, we turn to showing these completeness results. The following lemma implies Theorem 1.

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

**Proof.** Let’s see why \( \text{NSC}^k \subseteq \text{NL}[\log^{k-1} n] \) first. Let \( M \) be a machine that accepts a language \( L \in \text{NSC}^k \). 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 \( i \)th 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 \( i \)th pass, starting from the initial configuration, \( M' \) will check that the \( i \)th 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 \( i \)th block, then \( M' \) will merely check that \( i \)th blocks of the two configurations are identical; if the head is in the \( i \)th 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 \( i \)th block; in those cases, the blocks \( i - 1 \) and \( i + 1 \) will also need to be consulted. If the \( i \)th 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 \( \text{NL}[\log^{k-1} n] \) that is contained in \( \text{NSC}^k \). \( \text{SAT}_{\text{pw}}(\log^k |\phi|) \) is such a problem, by the following characterization:

**Lemma 3 (\cite{Pap99}).** \( \text{SAT}_{\text{pw}}(\log^k |\phi|) \) is complete for \( \text{NL}[\log^{k-1} n], \text{ for } k \in \mathbb{Z}^+, \text{ under log-space many-to-one reductions} \).

A nondeterministic machine \( M'' \) for \( \text{SAT}_{\text{pw}}(\log^k |\phi|) \) 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 \( j \)th 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. \( \mathcal{M}'' \) uses \( O(\log^k n) \) space and polynomial time.

Lemma 3 and Lemma 2 immediately yield the following theorem:

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

**Theorem 2.** \( \text{SAT}_{tw}(\log^k |\phi|) \) is complete for \( \text{SAC}^k_{\text{quasi}} \), 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.** \( \text{SAT}_{tw}(\log^k |\phi|) \in \text{NAuxPDA}(\log^k n, n^{O(1)}) \)

**Proof.** The algorithm witnessing this containment is very natural when expressed as a \( \text{NAuxPDA} \); 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 that \( \text{SAT}_{pw}(\log^k |\phi|) \) is in \( \text{NSC}^k \).

The \( \text{NAuxPDA} \) 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 \( \text{NAuxPDA} \) 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 \( \text{NAuxPDA} \) will first check if \( v \) has any children. If not, the \( \text{NAuxPDA} \) 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 \( \text{NAuxPDA} \) will guess assignments for \( v \)'s children (of which there are \( \leq 2 \)), 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 \( \text{NAuxPDA} \) will pop the information for the second child off of the stack and search it. If both subtrees are successfully searched, then the \( \text{NAuxPDA} \) 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 \( \text{SAC}^k_{\text{quasi}} \). Similar “generic reductions” (i.e. reducing the computation of families of \( \text{SAC} \) circuits) for treewidth-related problems have appeared before, e.g. [GLS01].

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

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

In particular, note that we may assume the following normal form for \( C \): (i) \( C \) is layered, (ii) \( C \) is
strictly alternating: odd-layer gates are OR, even-layer gates are AND, (iii) \( C \) has an odd number of layers, and (iv) the AND gates in \( C \) have fan-in 2.

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

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

A correct proof-tree exists if and only if, for each edge \((v, u)\), in the skeleton, the assignments to the variables in \( x_v, d_v \) and \( x_u \) can be picked so that \( M \) accepts the corresponding tuples. This condition can be formalized as \( \exists s, M'(s) = 1 \), where \( |s| = O(\log^k n) \), corresponding to the input bits provided to a Turing machine \( M' \) (a modification of \( M \)) having running time \( O(\log^k n) \) on \( s \). We would like to encode this à la Cook-Levin (see e.g. [AB09]) as a CNF of size \( O(\log^k n) \) – but
there is a catch. Using the tools provided in [AB09], this is only possible if \( M' \) is *oblivious* – and a naïve approach to making \( M' \) oblivious would introduce an unwanted \( \log \log n \) factor; thus we need to look more closely at the condition that \( M' \) is checking.

\( M' \) is taking \( s \) as input, and checking that \( s \) is giving information about adjacent gates in \( C \). Since all of the valid labels for a node in the skeleton are concerned with the same segment of an oblivious \( \text{NAuxPDA} \)’s computation, we can use the standard technique (e.g. [AB09]) to build a CNF of size \( O(\log^k n) \) verifying that the connectivity information is correct. (For example, \( s \) could give the encodings for gates labeled \((A, B)\) and \((C, D, E, \gamma)\), and we need to verify that the \( \text{NAuxPDA} \) can move from \( A \) to \( C \) pushing \( \gamma \), and move from \( E \) to \( B \) popping \( \gamma \).) At the end we take the conjunction of all the CNFs corresponding to the nodes and edges, which is also a CNF \( F \), where \( F \) is satisfiable if and only if \( C(x) = 1 \).

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

\( \square \)

**Remark 3.** In general, when the tree-width is anything larger than poly-logarithmic, the previous reductions still hold. In particular, \( \text{SAT}_{\text{tw}}(w(|\phi|)) \) is complete for \( \text{SAC}(\log |\phi|, 2^{O(w(|\phi|))}) \).

### 3.4 Connecting Conjecture [1] to complexity theory assumptions

and the separation of \( \text{SAT}_{\text{pw}}(\log^k |\phi|) \) from \( \text{SAT}_{\text{tw}}(\log^k |\phi|) \)

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

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

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

This corollary is just a resource-scaled form of our initial equivalence for logarithmic tree-width. In fact, by padding\(^3\) we have:

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

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

---

\(^3\) Philosophically, the assumption \( \text{SAC}_{\text{quasi}}^k \not\subseteq \text{TISP}(2^{O(\log^k n)}, n^{o(1)}) \) is not really different than the widely-believed assumption \( \text{SAC}^1 \not\subseteq \text{TISP}(n^{O(1)}, n^{o(1)}) \). By analogy let us consider \( \text{P} \not\subseteq \text{NP} \) and \( \text{E} \not\subseteq \text{NE} \). It is true that \( \text{E} \not\subseteq \text{NE} \) is stronger in the sense that \( \text{E} \not\subseteq \text{NE} \) implies \( \text{P} \not\subseteq \text{NP} \) (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 [BWMRS2]. 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 \( \text{SAC}^1 \not\subseteq \text{TISP}(n^{O(1)}, n^{o(1)}) \) for logarithmic tree-width, whereas for larger tree-width we have only shown equivalence to the scaled analogs of \( \text{SAC}^1 \not\subseteq \text{TISP}(n^{O(1)}, n^{o(1)}) \).
i.e. validating our conjecture for restricted types of algorithms implies progress towards $\text{NC} \neq \text{SC}$.

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

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

In fact, the above holds up to $\text{NL}$-reductions. This corollary extends to every poly-logarithmic width under the scaled assumption $\text{NSC}^k \subset \text{SAC}^k_{\text{quasi}}$. This is the first separation result for width parameterizations of $\text{SAT}$ 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 $(2^{2^{TW(\phi)}} |\phi|^{O(1)}, 2^{TW(\phi)} |\phi|^{O(1)})$, whereas the space-efficient one works in time-space $(3^{TW(\phi)} \log |\phi| |\phi|^{O(1)}, |\phi|^{O(1)})$. 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. Later on, in Section 4.1, we provide a primer to algorithms for $\text{SAT}$ instances with given tree decompositions.

**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 $2^{TW(\phi)}$ size Boolean array; entry $j$ in the array will be 1 if the subformula rooted at the bag is satisfied, when the variables are given assignment $j$, and will be 0 otherwise. Clearly, computing the array for the root will solve the satisfiability of the formula, and indeed by the property of a tree decomposition, the array values can be computed in a leaves-to-root fashion.

To simplify the overview of the space-efficient algorithm we shall temporarily assume that each clause appears in a bag together with all of its variables. Observe that if we fix a truth assignment on a bag, then solving $\text{SAT}$ on the given tree decomposition reduces to solving e.g. 3 independent subproblems – think of splitting the degree-3 tree into three subtrees by cutting the original one at this bag. The algorithm works by recursively enumerating and checking truth assignments on the bags. Its performance is determined by the size of the subproblems (ideally all the subtrees have the same size). In Section 4.2 (Lemma 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 4.2 are a bit of an overkill for the analysis of this simple algorithm, but they are also applied in the analysis of the tradeoff.

**The tradeoff algorithm: where is the complication?** Let us consider for a moment an execution of the space-bounded algorithm. We can visualize each step of the recursion as splitting the tree decomposition at a node (bag) – this bag is replicated at each of the subproblems with the

---

3We remove this assumption later; see Section 4.1
fixed truth assignment. Let the process evolve for a while, and when the forest has enough trees let us single out one such tree. At the boundary (the leaves) of this tree there can be as many as \( \log |\phi| \) nodes to which we previously fixed an assignment, i.e. by splitting. The logarithmically large number of nodes does not affect the performance of the space-efficient algorithm (at each point of the recursion each bag/node is associated with a single assignment). Now, we switch gears to devise a tradeoff algorithm. A natural thing to do is first to discretize the truth assignment space associated with each bag, say in \( 2^{(1-\epsilon)TW(\phi)} \) many chunks each of size \( 2^{\epsilon TW(\phi)} \), 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 \( 2^{TW(\phi)} \) down to \( 2^{(1-\epsilon)TW(\phi)} \). 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 \( 2^{\epsilon \log |\phi|TW(\phi)} \). Overall this is a time-space \( (2^{(1-\epsilon) \log |\phi|TW(\phi)}, 2^{\epsilon \log |\phi|TW(\phi)}) \) algorithm, worse both than the time- and space-efficient ones! To devise our tradeoff algorithm we show that it is possible to simultaneously (i) perform the splitting in a way that at each step of the execution the forest consists of trees each with at most a constant number of split-nodes and (ii) this splitting results in subproblems of somewhat balanced sizes. Furthermore, we show that it is possible to control the number of splitting nodes per tree in the forest in a way that yields a tradeoff on this parameter (Section 5). This is a different (and competing) tradeoff from the one by the discretization factor \( \epsilon \); i.e. our most general tradeoff algorithm is controlled by two parameters.

4.1 A primer to algorithms for width-parameterized SAT

The structure of a tree decomposition is associated with the concept of separability (see e.g. [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. Recall that, for this initial overview, we are assuming that all the variables of a clause appear in the same bag with the clauses. We will see later that removing this assumption in a time-efficient manner is non-trivial (in fact, removing it without increasing the base of the exponential running time is an interesting puzzle).

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

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

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

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

![Figure 3](image-url)  
Figure 3: Three instances used in the example. Figures on the top are the input tree decompositions, the bottom figures are the two components after fixing assignment to the variables in the middle bag.

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

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

Satisfying $C_5$ requires $x_4 = 0$, which implies that $C_1$ can not be satisfied by $x_4$. Similarly, satisfying $C_6$ requires $x_5 = 0$, so that $C_2$ can not be satisfied by $x_5$. In order to find a satisfying truth assignment, when processing the right part, we need information about which of $C_1, C_2$ is already satisfied in the left part. (Since $\phi_1$ is not satisfiable, the final outcome will be the same in either case.) $\phi_2$ is satisfied only when $C_1$ is already satisfied, while $\phi_3$ is satisfied only when $C_2$ is already satisfied. This piece of information is not carried through the middle bag by just the truth assignment to the variables. To overcome this issue we are going to use “clause-bits”, which we mentioned briefly in Section 2.3.

### 4.2 Splitting, Consistency, Assignment Groups

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

**Definition 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. 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 $T_i$, $1 \leq i \leq k$.

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

A splitting algorithm $A$ computes a function that, given a tree $T$ 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 $0 < \alpha < 1$, and satisfying $N > 1/\alpha$. Then, there is a node $p$ where after we split at $p$, the tree which contains $s$ is of size $\leq \lceil \alpha N \rceil$ and every other tree is of size $\leq \lceil (1 - \alpha)N \rceil$. The node $p$ is called an $\alpha$-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 $(s \equiv v_1, v_2, \ldots, v_l)$ 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 $\alpha$-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 $\alpha N$. 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 $X^* = \cup_{i \in S} X_i$, and let $V$ be the variables and $C$ 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, $T$ splits into subtrees $T_i$’s. In every splitting operation, the bag being split is given some assignment. Further suppose $R_T$ is an assignment to $T$, and $R_{T_i}$ is an assignment to the subtree $T_i$. $R_T$ and $R_{T_i}$’s are said to be consistent if

1. for every $i$, the bits corresponding to a variable $x$ in $R_{T_i}$ is the same as in $R_T$
2. for a clause $C$:
   a) if $C$ appears in $X^*$ and is assigned to 0, then $\forall i$ every bit for $C$ in $R_{T_i}$ is assigned to 0
   b) otherwise, $\exists$ exactly one $i$ such that in $R_{T_i}$ the bit corresponding to $C$ is assigned to 1.

Remark 4. The latter point in the definition, where in exactly one of the subtrees we require that the corresponding bit equals to 1, is somewhat subtle. The following lemma crucially depends on this property.

Lemma 7. For every assignment $R_T$ to the tree $T$, the number of assignments $R_{T_i}$ to subtrees $T_i$’s consistent with $R_T$ is at most $d^{TV(\phi)}$. (Recall $d \in \{2,3\}$.)

Proof. Let $X_p$ be the bag corresponding to the splitting node $p$. For each variable $x$ in the bag $X_p$, there are 2 possible assignments of the bit for $x$ in the $T_i$’s. For each clause $C$ in $X_p$, if $C$ appears
Consider splitting at the gray bag, while fixing the value of the bits, 1 for $C_1$, 0 for $x_3$. Any possible consistent assignments for $T_1, T_2, T_3$ will have 0 for $x_3$; in this consistent assignment $C_1$ has value 1 in $T_1$, and has value 0 in $T_2$ and $T_3$.

in $R_T$ and is assigned to 0, by the definition of consistency, all the bits for $C$ in the $T_i$'s are assigned to 0. Otherwise, in exactly one $T_i$, the bit for $C$ is assigned to 1; in this case there are at most $d$ valid assignments. Recall that $d$ is the maximum degree of the tree decomposition.

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

**Definition 4.** For a tree $T$ with splitting nodes $S$, an assignment $R_T$ is **satisfying** if there exists a truth assignment $A$ to every variable in $T$, such that

(1) every truth value for a variable in $R_T$ agrees with the corresponding value in $A$

(2) every clause $C$ that appears in $T$ 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 $R_T$ 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 $R_T$ is satisfying if and only if there exist satisfying assignments $R_{T_i}$ to each subtree $T_i$, such that all of the assignments $R_{T_i}$ are consistent with $R_T$.

**Proof.** For a tree $T$ with splitting nodes $S$, suppose that splitting at node $p$ results in the subtrees $\{T_i\}$. Suppose that the assignment $R_T$ is satisfying. By Definition 4 there exists a truth assignment on variables within $T$ that makes all of the clauses true. This truth assignment induces assignments $R_{T_i}$ consistent with $R_T$, such that for these truth assignments the conditions in Definition 4 are met. (Some of the clause bits in the $R_{T_i}$ may need to be set to zero, to ensure consistency.)

For the other direction suppose that there exist assignments $R_{T_i}$ of the subtrees $T_i$, such that the assignments $R_{T_i}$'s are consistent with $R_T$ and all $R_{T_i}$'s are satisfying. For each subtree $T_i$, there
exists a truth assignment complying to Definition 4. Since all these truth assignments agree on their common variables (because the common variables appear in splitting nodes in $S$), we can get a truth assignment from their union, which also meets the axioms in Definition 4. Therefore, the assignment $R_T$ is satisfying.

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

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

Note that the fixed entries for the splitting node $p$ may be different among subtrees (because of the rules regarding clause bits), and note also that some of the unfixed entries in $T$ may fixed in subtrees (because they appear in $p$). The following important lemma holds, which basically generalizes Lemma 7.

**Lemma 9.** The number of distinct $\varepsilon$-GR$_{T_i}$'s consistent with $\varepsilon$-GR$_T$ is at most $d(1 - \varepsilon)^{TW(\phi)}$.

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

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

2. $C$ is in some previous splitting node, and its value is fixed in $\varepsilon$-GR$_T$. If the bit for $C$ is assigned 1, then there are $d_0$ possible assignments to $C$ similar as above, otherwise the only possible way is to set all bits for $C$ to 0.

Since there are at most $(1 - \varepsilon)TW(\phi)$ entries that need to be fixed, in order to form the $\varepsilon$-GR$_{T_i}$'s, the number of different combinations of $\varepsilon$-GR$_{T_i}$'s consistent with $\varepsilon$-GR$_T$ is at most $d(1 - \varepsilon)^{TW(\phi)}$.
4.3 The space-efficient algorithm

The space-efficient algorithm is described in Algorithm 1. \( T \) is a tree with previous splitting nodes \( S \), and \( R_T \) is the assignment fixed on the tree. A subtle point that affects the running time of this algorithm is addressed in Remark 4. The correctness of the algorithm directly follows by Lemma 8.

Algorithm 1 SAT(\( T \), \( R_T \))

1: if all nodes in \( T \) are previous splitting nodes then
2:     if every clause in \( R_T \) assigned value 1 is satisfied by some variable in \( T \) then
3:         return True
4:     else
5:         return False
6: end if
7: else
8:     split at the 1/2-splitting node, and denote the subtrees as \( T_i \)'s
9:     for all \( R_T \)'s consistent with \( R_T \) do
10:        if \( \forall T_i, \text{SAT}(T_i, R_{T_i}) = \text{True} \) 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

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

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

4.4 Tradeoff Algorithms

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

A type_\( \ell \) tree is a tree with \( \ell \) previous splitting nodes. Let \( \alpha \) 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 a type_\( \ell \) tree, for any \( \ell \geq 3 \).

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.
Algorithm 2 SAT-tradeoff(\( \mathcal{T}, \mathcal{S}, \varepsilon\text{-GR}_T \))

1: \( M(\mathcal{T}, \varepsilon\text{-GR}_T) \leftarrow \) all-zero array
2: if all nodes in \( \mathcal{T} \) are in \( \mathcal{S} \) then
3:     \( \) for all \( j : 1 \leq j \leq |\varepsilon\text{-GR}_T| \) do
4:         \( R_T \leftarrow j \)th assignment in \( \varepsilon\text{-GR}_T \)
5:     \( \) if every clause whose bit in \( R_T \) assigned 1 is satisfied by some variable in \( \mathcal{T} \) then
6:         \( M(\mathcal{T}, \varepsilon\text{-GR}_T)_j \leftarrow 1 \)
7:     \( \) end if
8: \( \) end for
9: else
10: \( \) split at Splitting Alg(\( \mathcal{T}, \mathcal{S} \)), and denote the subtrees \( T_i \)'s \( \) Replaceable
11: \( \) for all \( \varepsilon\text{-GR}_T \)'s consistent with \( \varepsilon\text{-GR}_T \) by fixing \((1 - \varepsilon)TW(\phi)\) entries do
12: \( \) \( \forall i, M(\mathcal{T}_i, \varepsilon\text{-GR}_{T_i}) \leftarrow \) SAT-tradeoff(\( \mathcal{T}, \mathcal{T}_i, \varepsilon\text{-GR}_{T_i} \))
13: \( \) for all \( j : 1 \leq j \leq |\varepsilon\text{-GR}_T| \) do
14: \( \) \( R_T \leftarrow j \)th assignment in \( \varepsilon\text{-GR}_T \)
15: \( \) \( \) for all \( R_T \)'s chosen \( \varepsilon\text{-GR}_{T_i} \)'s correspondingly do
16: \( \) \( \) if \( \forall i, M(\mathcal{T}_i, R_T) = 1 \) and \( \forall T_i, R_T \)'s are consistent with \( R_T \) then
17: \( \) \( \) \( M(\mathcal{T}, \varepsilon\text{-GR}_T)_j \leftarrow 1 \)
18: \( \) end if
19: \( \) end for
20: \( \) end for
21: \( \) end if
22: \( \) end if
23: \( \) return \( M(\mathcal{T}, \varepsilon\text{-GR}_T) \)

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

Proof. Denote by \( T_1(N) \), \( T_2(N) \) the running time of Algorithm 3 using splitting rule \( \mathcal{H}_2 \) on type 1 or type 2 trees each of \( N \) nodes respectively. Splitting a type 1 tree results in multiple type 1 trees with size at most \((1 - \alpha)N\) and one type 2 tree with size at most \( \alpha N \), so we have

\[
T_1(N) \leq O(d^{(1-\varepsilon)TW(\phi)})(T_1((1 - \alpha)N) + T_2(\alpha N)) + 2^{O(TW(\phi))}
\]

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

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

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

\[
T_1(N) \leq O(d^{(1-\varepsilon)TW(\phi)})(T_1((1 - \alpha)N) + T_2(\alpha N)) + 2^{O(TW(\phi))}
\]
\[
\leq O(d^{(1-\varepsilon)TW(\phi)})T_1((1 - \alpha)N) + O(d^{2(1-\varepsilon)TW(\phi)})T_1(\alpha N) + 2^{O(TW(\phi))}
\]

Therefore:

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

23
Algorithm 3 $H_2(\mathcal{T}, S)$

1: if $\mathcal{T}$ with $S$ is a type 0 tree then
2: return the $1/2$-splitting node
3: else if $\mathcal{T}$ with $S$ is a type 1 tree then
4: regard the previous splitting node as root
5: return the $\alpha$-splitting node $\Delta$ Figure 6a
6: else if $\mathcal{T}$ with $S$ is a type 2 tree then
7: suppose $S = \{p_1, p_2\}$
8: regard $p_1$ as root and compute the $1/2$-splitting node $q$
9: if $q$ is on the path between $p_1$ and $p_2$ then
10: return $q$ $\Delta$ Figure 6b
11: else
12: return the least common ancestor of $q$ and $p_2$ $\Delta$ Figure 6c
13: end if
14: end if

Since type $i$, $i \geq 3$ trees are not allowed, the space requirement is $2^{2cTW(\phi)}|\phi|O(1)$ □

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

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

Definition 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 3 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 a root node \( r \) and the root of subtree \( F_h \).

![Figure 7: Fibonacci tree and extended Fibonacci tree](image)

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

**Theorem 4.** Every algorithm in \( \mathfrak{A}_2 \) runs in \( \Omega(3^{1.441(1-\varepsilon)}TW(\phi)^{\log N}|\phi|^{\Theta(1)}) \) time on \( F^*_{h+2,r} \).

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

**Lemma 10.** Every algorithm in \( \mathfrak{A}_2 \) runs in \( \Omega(3^{(1-\varepsilon)}TW(\phi)^{h+1}|\phi|^{\Theta(1)}) \) time on \( \mathcal{T}_{1,h} \), and runs in \( \Omega(3^{(1-\varepsilon)}TW(\phi)^{(h+1)}|\phi|^{\Theta(1)}) \) on \( \mathcal{T}_{2,h} \).

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

(1) For \( \mathcal{T}_{1,h} \), if we split at the root of \( F_h \), a \( \mathcal{T}_{1,h-1} \) will be derived, the running time on which is \( \Omega(3^{(1-\varepsilon)}TW(\phi)^{(h-1)}|\phi|^{\Theta(1)}) \); otherwise if we split at some node inside the subtrees \( F_{h-1} \) or \( F_{h-2} \), the running time on the subtree containing two splitting nodes (a new one and a previous one) is lower bounded by the running time of a \( \mathcal{T}_{2,h-2} \), which is \( \Omega(3^{(1-\varepsilon)}TW(\phi)^{(h-2)+1)}|\phi|^{\Theta(1)}) \). To see this, assume for example that the splitting node \( p \) is inside the \( F_{h-1} \). Imagine that all the nodes in \( F_{h-1} \) except those on the path between \( p \) and the node \( v \) which connects to the root of \( F_{h-2} \) are pruned for free, and then replace the path between \( p \) and \( v \) by an edge between them. A \( \mathcal{T}_{2,h-2} \) is derived this way, and note that the running time will not increase after the procedure. Observe that enumerating all assignments for a newly created splitting node requires time \( \Omega(3^{(1-\varepsilon)}TW(\phi)^{h}|\phi|^{\Theta(1)}) \), so the running time for \( \mathcal{T}_{1,h} \) is \( \Omega(3^{(1-\varepsilon)}TW(\phi)^{h}|\phi|^{\Theta(1)}) \).
(2) For $T_{2,h}$, the splitting must occur at the node connecting the two splitting nodes, which creates a $T_{1,h}$, the running time on which as shown above is $\Omega(3^{1-\varepsilon}TW(\phi)h|\phi|\Theta(1))$. Taking the time required for enumerating all assignments for the new splitting node into consideration, the total running time for $T_{2,h}$ is therefore lower bounded by $\Omega(3^{1-\varepsilon}TW(\phi)(h+1)|\phi|\Theta(1))$.

Proof of Theorem 4. The very first step of splitting at any node of $F_{h+2,r}$ will result in a type 1 tree, the running time on which is lower bounded by the running time on $T_{1,h}$. By Lemma 10, that is $\Omega(3^{-\log(1-\varepsilon)}(1-\varepsilon)TW(\phi)\log N|\phi|\Theta(1))$, where $N = \Theta((1+\sqrt{5})h)$, and $\alpha = \frac{3-\sqrt{5}}{2}$ (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 $(\varepsilon, c)$. Moreover, we show that both of these parameters are necessary to achieve different tradeoff algorithms. 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 $\varepsilon$, where $0 < \varepsilon < 1$, a SAT instance $\phi$ with a tree decomposition of width $TW(\phi)$ and $N$ nodes, can be decided in time-space $(3^{1-\log(1-\varepsilon)}(1-\varepsilon)TW(\phi)\log N|\phi|\Theta(1))$, where $N = \Theta((1+\sqrt{5})h)$, and $\alpha = \frac{3-\sqrt{5}}{2}$ (which matches the parameter chosen in the tradeoff algorithm). By simplifying this expression we obtain the theorem.

$\lambda_c$ is a constant depending on $c$. To be more specific, $\lambda_c$ 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 $\lambda_c$ for small $c$’s are listed in Table 1.

<table>
<thead>
<tr>
<th>$c$</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\lambda_c$</td>
<td>1.441</td>
<td>1.138</td>
<td>1.057</td>
<td>1.026</td>
<td>1.013</td>
</tr>
</tbody>
</table>

Table 1: Values of $\lambda_c$ for small $c$’s

5.1 Generalized tradeoff algorithms

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

First, we generalize the splitting algorithm to allow type $i$ trees for $i$ up to $c$. For arbitrary $1 \leq i \leq c$, 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 $H_c$, such that when splitting a $type_i$ tree, we invoke $H_c$ to determine the splitting node. This is an implementation of Splitting Alg in Algorithm 2.

**Algorithm 4** $H_c(T, S)$

1. if $T$ with $S$ is a $type_0$ tree then
2. return the 1/2-splitting node
3. else
4. suppose $T$ with $S$ is a $type_i$ tree
5. if $|T| \leq 2^{c-i}$ then
6. return the 1/2-splitting node
7. else
8. pick an arbitrary node from $S$ as root and compute the $\alpha_{c,i}$-splitting node $q_1$
9. if $q_1$ is not on the path between any pair in $S$ then
10. return $q_1$ ▶ Figure 8a
11. else
12. compute a 1/2-splitting node $q_2$.
13. if $q_2$ is on the path between any pair in $S$ then
14. return $q_2$ ▶ Figure 8b
15. else
16. return the least common ancestor of $q_2$ and all nodes in $S$ ▶ Figure 8c
17. end if
18. end if
19. end if
20. end if

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 node of a $type_c$ tree must be on the path between some pair of previous splitting nodes, this is assured by setting $\alpha_{c,c} = 0$. For a fixed $c$, the running time and space of the algorithm solving SAT of bounded tree-width utilizing the splitting algorithm $A$ are summarized in Theorem 5 (see page 26).

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

**Definition 7.** The $c$-splitting depth $SD_c(A, T, S)$ of a splitting algorithm $A$ on tree $T$ with previous splitting nodes $S$ is inductively defined as follows (where the case for $|S| > c$ is arbitrary):

$$
SD_c(A, T, S) = \begin{cases} 
\max_{(T_0, S_0) \in C_{T,S,p}} SD_c(A, T_0, S_0) + 1 & |S| \leq c, |S| < |T| \\
0 & |S| \leq c, |S| = |T| \\
\infty & |S| > c 
\end{cases}
$$

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

We define the $c$-minimum splitting depth $\min SD_c(T, S)$ to be the minimum value of $SD_c(A, T, S)$, over all splitting algorithms.

Under this notation, given a tree $T$, any algorithm $A$ avoiding $type_{c+1}$ trees requires time $d(1-\varepsilon)^{SD_c(A, T, S)} |T| W(\phi) O(1)$ and space $2^{c-\varepsilon} T W(\phi) |\phi| O(1)$. 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
two technical lemmas: Lemma 11 establishes the recurrences according to the recursive algorithm, and Lemma 12 deals with choice of parameters. For simplicity of presentation we ignore issues regarding the divisibility of $N$ by 2.

**Lemma 11.** For every $c \geq 2$, any tree $T$ with $N$ nodes and splitting node set $S$ of size $i$, let $D_{c,i}(N) = \max_{T,S} SD_c(H_c, T, S)$. Then,

\[
\begin{align*}
D_{c,i}(N) &\leq \max\{D_{c,1}((1 - \alpha_{c,i})N), D_{c,i+1}(\alpha_{c,i}N), D_{c,i}(N/2)\} + 1 & \forall i : 1 \leq i < c \\
D_{c,c}(N) &\leq \max\{D_{c,1}(N), D_{c,c}(N/2)\} + 1
\end{align*}
\]

**Proof.** Without loss of generality, suppose $N \geq 2^c$. Consider splitting a type$_i$ tree with splitting nodes $S$, $1 \leq i < c$. If the $\alpha_{c,i}$-splitting-node $m$ is not on the path between any pair of previous splitting nodes, splitting at $m$ will result in multiple type$_1$ trees of size at most $\lceil (1 - \alpha_{c,i})N \rceil$ and one type$_{i+1}$ tree of size at most $\lceil \alpha_{c,i}N \rceil$. Otherwise, since $1 - \alpha_{c,i} \geq 1/2$, the maximal possible size of a type$_1$ tree created by any splitting node will not exceed $\lceil (1 - \alpha_{c,i})N \rceil$. Splitting at the 1/2-splitting-node $c$ will result in multiple type$_j(j \leq i)$ trees of size at most $\lceil N/2 \rceil$, otherwise, splitting at the least common ancestor of $c$ and all previous splitting nodes as $p$, will result in multiple type$_1$ tree of size at most $\lceil (1 - \alpha_{c,i})N \rceil$ and many type$_j(j \leq i)$ trees with size at most $\lceil N/2 \rceil$. In summary,

\[
D_{c,i}(N) \leq \max\{D_{c,1}((1 - \alpha_{c,i})N), D_{c,i+1}(\alpha_{c,i}N), D_{c,i}(N/2)\} + 1
\]

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 12. SD$_c(H_2, \mathcal{T}, \emptyset)$ for a tree $\mathcal{T}$ of $N$ nodes is at most $\lambda_c(\log N - c) + c + O(1)$, with properly chosen parameters $\alpha_{c,i}$’s.

Proof. For ease of computation, we define a function $D'$ independent of $D$, as a recurrence that satisfies the recursion that, by Lemma 11, holds for the running time.

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

By manipulating the first equation, we can derive that

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

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,

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

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

$$\alpha_{c,i} = 1 - \frac{\alpha_{c,1}(1 - \alpha_{c,i})}{2\alpha_{c,i-1} - (1 - \alpha_{c,i})}, \quad \forall i : 1 \leq i \leq c$$

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

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

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

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

Note that $D'$ is defined in a way that it satisfies the recursion in Lemma 11 and it can be proved by induction that $D'_{c,i}(N)$ upper bounds $D_{c,i}(N)$ for all $c, i$. By the choice of the parameters, the recurrence can be solved using standard tools. Specifically,

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

Since $D_{c,1}(2^c) = c$, SD$_c(H_2, \mathcal{T}, S)$ is upper bounded by $\lambda_c(\log N - c) + c + O(1)$, where $\lambda_c$ satisfies $\sum_{i=1}^{c} 2^{-\lambda_c} = 1$. \hfill \Box
Proof of Theorem 5. For every $c \geq 2$, we solve the above recurrences: when $N < 2^c$, the running time is $d^{\log N (1-\varepsilon) TW(\phi)} |\phi|^{O(1)}$; when $N \geq 2^c$, the running time is $d^{(\lambda_c (\log N - c) + c (1 - \varepsilon)) TW(\phi)} |\phi|^{O(1)}$. Space required by the algorithm is upper bounded by $2^{\varepsilon TW(\phi)} |\phi|^{O(1)}$ when only $\lambda_i$ trees are allowed, for $i \leq c$.

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

**Lemma 13.** $\lambda_c < 1 + \frac{2}{2^{c/2}}$

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

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

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

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 $2^{\varepsilon' TW(\phi)} |\phi|^{O(1)}$ and time $d^{\delta TW(\phi) \log |\phi| O(1)}$ for a constant $\delta < 1$.

**Proof.** For any fixed $\varepsilon$ and $c$, there is an algorithm with running time $O(d^{\lambda_c (1 - \varepsilon) TW(\phi) \log |\phi| O(1)})$ and space $O(2^{\varepsilon' TW(\phi)} |\phi|^{O(1)})$ by Theorem 5. Set $\varepsilon = \frac{\varepsilon'}{c}$, then the space is $O(2^{\varepsilon TW(\phi)} |\phi|^{O(1)})$ and the running time is $O(d^{\lambda_c (1 - \varepsilon') TW(\phi) \log |\phi| O(1)})$. By Lemma 13, $\lambda_c (1 - \varepsilon') < (1 + \frac{2}{2^{c/2}}) (1 - \varepsilon') < 1$ for sufficiently large $c$.

### 5.2 Optimality of the generalized tradeoff algorithm

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

**Definition 8.** For any integer $c \geq 2$, and positive integer $h$, a $(c,h)$-Fibonacci tree (denoted as $F_{c,h}$) is a rooted tree defined by one of the rules,

1. if $h \leq c$, $F_{c,h}$ is a chain of $2^c$ nodes;
2. if $h > c$, $F_{c,h}$ is constructed by starting from a chain of $c$ nodes (one end as the root), then replacing the $i$th 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 $F_{c,h}$. 

30
See Figure 9a for an illustration of a \((c, h)\)-Fibonacci tree. \(F_{c,h+c}\) is indeed the hard instance for splitting algorithms in \(\mathcal{A}_c\) for \(c \geq 2\). As discussed previously, it suffices to lower bound \(\text{minSD}^c\) on this tree.

![Fibonacci tree in a more general form and the hard instance based on it](image)

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

Similar to the proof of Theorem 3, we need a definition of trees which may appear in intermediate phases and contain previous splitting nodes. \(\forall c \geq 2, \forall h > c\) and \(\forall w : 1 \leq w \leq c\), \(G_{c,h,w}\) is such a tree: first construct a chain of length \(w\), then connect \(c - w + 1\) previous splitting nodes to the first node of the chain, and finally shrink the path between the first node on the chain and the splitting node, then shrink the first edge on the chain to obtain a \(G_{c,h}\) outside the path between the first node on the chain and the splitting node, this produces a \(G_{c,h} = G_{c,h-1}\) splitting at the first node of the chain. By induction hypothesis (on \(h\) holds for any \(F\) the splitting node is inside the subtree procedure which will not increase the running time will be applied to the tree after splitting. If \(h > c\) then \(\text{minSD}^c(\mathcal{F}_{c,h}, \mathcal{S}_{c-w+1}) \geq h - c + w\) holds for any proper \(w\). For the induction step, we prove

\[
\text{minSD}^c(\mathcal{F}_{c,h_0}, \mathcal{S}_{c-w+1}) \geq h_0 - c + w
\]

holds for any \(w\) by induction on \(w\). When \(w = 1\), to prevent type\(_i\) trees for \(i > c\), we must split at the first node of the chain. By induction hypothesis (on \(h\)), \(\text{minSD}^c(\mathcal{F}_{c,h_0-1}, \mathcal{S}_c) \geq 1 + \text{minSD}^c(\mathcal{F}_{c,h_0-c}, \mathcal{S}_1) \geq h_0 - c + 1\). Suppose \(3\) holds \(\forall w : w < w_0\), for the induction step, consider lower bounding \(\text{minSD}^c(\mathcal{F}_{c,h_0-1}, \mathcal{S}_{c-w+1})\).

As in the proof of Lemma 10, in order to apply the induction hypothesis, some size-reducing procedure which will not increase the running time will be applied to the tree after splitting. If the splitting node is inside the subtree \(F_{c,h_0-c+w_0-1}\), prune the nodes in the subtree \(F_{c,h_0-c+w_0-1}\) outside the path between the first node on the chain and the splitting node, then shrink the path to an edge, and finally shrink the first edge on the chain to obtain a \(G_{c,h_0,w_0-1}\), by induction hypothesis, \(\text{minSD}^c(\mathcal{F}_{c,h_0,w_0-1}, \mathcal{S}_{c-(w_0-1)+1}) \geq h_0 - c + w_0 - 1\); otherwise consider an additional free splitting at the first node on the chain, this produces a \(G_{c,h_0-c+w_0-1,c}\), again by induction hypothesis \((h_0 - c + w_0 - 1 < h_0)\), \(\text{minSD}^c(\mathcal{F}_{c,h_0-c+w_0-1,c}, \mathcal{S}_1) = h_0 - c + w_0 - 1 - c + c = h_0 - c + w_0 - 1\). Therefore, \(\text{minSD}^c(\mathcal{F}_{c,h_0,w_0}, \mathcal{S}_{c-w_0+1}) \geq 1 + h_0 - c + w_0 - 1 = h_0 - c + w_0\). This completes both inductions and also the proof.

**Corollary 6.** For each \(h \geq 1\), \(\text{minSD}(\mathcal{F}^*, \{r\}) \geq h\).
Proof. By Lemma [14], \( \minSD_c(F_{c,h,r}^*, \{r\}) = \minSD_c(G_{c,h,c}, S_1) \geq h - c + c \geq h. \)

Now we are ready to state the theorem of optimality,

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

**Proof.** Let \( |F_{c,h}| \) be the number of nodes in the tree \( F_{c,h} \). For any \( h \leq c \), we have \( |F_{c,h}| \leq 2^c \), when \( h > c \), we have \( |F_{c,h}| = \sum_{i=1}^c |F_{c,h-i}| + c. \) By the recursion, the generating function of \( |F_{c,h}| \) can be written as \( f(X) = X^c - \sum_{i=0}^c X^i. \) Therefore \( |F_{c,h}| = \sum_{i=1}^c \delta_{c,i} \gamma_{c,i}^{h-c} \), where \( \delta_{c,i} \) is upper bounded by a constant times \( 2^c \) and \( \gamma_{c,i} \) is the \( i \)-th root of the equation \( f(X) = 0. \)

Let \( \gamma_c = \arg \max_i \{|\gamma_{c,i}|\} \). When \( h \) tends to infinity, \( |F_{c,h}| = \Theta(2^c \gamma_{c,h-c}) \). So, \( h \geq \log \gamma_c (|F_{c,h}|/2^c) + c - O(1) \). Therefore, for any \( c \geq 2 \) and large enough \( N \), consider the tree \( T = F_{c,h+c} \) with \( \geq N \) nodes. Splitting it at any node will result in a tree, whose minimum splitting depth is lower bounded by \( \minSD(F_{c,h,r}^*, \{r\}) \): if the splitting node is on the chain, then let \( r \) be the splitting node; otherwise let \( r \) be any node on the chain other than the one which is connected to the root of the subtree where the splitting node lay. (One can think of as splitting at \( r \) for free.) This, when combining with Corollary [6], completes the proof. 

Similarly to Theorem [4] here we conclude the optimality of our tradeoff algorithm. That is, for fixed \( c \geq 2 \), \( \forall \varepsilon : 0 < \varepsilon < 1 \), any algorithm in \( \mathfrak{A}_c \) there is an instance \( \phi \), 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-Impagliazo [BBI11] took the first step towards this direction. Such results can be also understood as partial progress towards \( \text{SC} \neq \text{NC}. \)

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. This work was also supported in part by the National Basic Research Program of China Grant 2011CBA00300, 2011CBA00301, the National Natural Science Foundation of China Grant 61033001, 61061130540, 61073174.

## References


<table>
<thead>
<tr>
<th>Year</th>
<th>Author(s)</th>
<th>Title</th>
<th>Journal</th>
<th>Volume/Issue</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>1986</td>
<td>N. Robertson and P.D. Seymour</td>
<td>Graph minors. II. algorithmic aspects of tree-width.</td>
<td>Journal of algorithms</td>
<td>7(3)</td>
<td>309–322</td>
</tr>
<tr>
<td>2010</td>
<td>M. Samer and S. Szeider</td>
<td>Algorithms for propositional model counting.</td>
<td>J. Discrete Algorithms</td>
<td>8(1)</td>
<td>50–64</td>
</tr>
<tr>
<td>1987</td>
<td>H. Venkateswaran</td>
<td>Properties that characterize LOGCFL.</td>
<td>Symposium on Theory of Computing (STOC)</td>
<td></td>
<td>141–150</td>
</tr>
<tr>
<td>1999</td>
<td>H. Vollmer</td>
<td>Introduction to Circuit Complexity.</td>
<td>Springer</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>