Solving #SAT and MaxSAT by dynamic programming

Sigve Hortemo Sæther, Jan Arne Telle, and Martin Vatshelle
{sigve.sether,telle,martin.vatshelle}@ii.uib.no

Department of Informatics, University of Bergen, Norway

Abstract. We look at dynamic programming algorithms for propositional model counting, also called #SAT, and MaxSAT. Tools from graph structure theory, in particular treewidth, have been used to successfully identify tractable cases in many subfields of AI, including SAT, Constraint Satisfaction Problems (CSP), Bayesian reasoning, and planning. In this paper we attack #SAT and MaxSAT using similar, but more modern, graph structure tools. The tractable cases will include formulas whose class of incidence graphs have not only unbounded treewidth but also unbounded clique-width. We show that our algorithms extend all previous results for MaxSAT and #SAT achieved by dynamic programming along structural decompositions of the incidence graph of the input formula. We present some limited experimental results, comparing implementations of our algorithms to state-of-the-art #SAT and MaxSAT solvers, as a proof of concept that warrants further research.

1 Introduction

The propositional satisfiability problem (SAT) is a fundamental problem in computer science and in AI. Many real-world applications such as planning, scheduling, and formal verification can be encoded into SAT and a SAT solver can be used to decide if there exists a solution. To decide how many solutions there are, the propositional model counting problem (#SAT), which finds the number of satisfying assignments, could be useful. If there are no solutions, it may be interesting to know how close we can get to a solution. When the propositional formula is encoded in Conjunctive Normal Form (CNF) this may be solved by the maximum satisfiability problem (MaxSAT), which finds the maximum number of clauses that can be satisfied by some assignment. In this paper we investigate classes of CNF formulas where these two problems, #SAT and MaxSAT, can be solved in polynomial time. Tools from graph structure theory, in particular treewidth, have been used to successfully identify tractable cases in many subfields of AI, including SAT, Constraint Satisfaction Problems (CSP), Bayesian reasoning, and planning, see for example [3,13,14,32]. In this paper we attack #SAT and MaxSAT using similar, but more modern, graph structure tools. The tractable cases will include formulas whose class of incidence graphs have not only unbounded treewidth but also unbounded clique-width.

Both #SAT and MaxSAT are significantly harder than simply deciding if a satisfying assignment exists. #SAT is #P-hard [17] even when restricted to Horn 2-CNF formulas, and to monotone 2-CNF formulas [30]. MaxSAT is NP-hard even when restricted to Horn 2-CNF formulas [20], and to 2-CNF formulas where each variable appears at most 3 times [27]. Both problems become tractable under certain structural restrictions obtained by bounding width parameters of graphs associated with formulas, see for example [14,16,32,34]. The work we present here is inspired by recent results of Paulusma et al [25] and Slivovsky and Szeider [33] showing that #SAT is solvable in polynomial time when the incidence graph[1] $I(F)$ of the input formula F has bounded modular treewidth, and more strongly, bounded symmetric clique-width.

These tractability results work by dynamic programming along a decomposition of $I(F)$. There are two steps involved: (1) find a good decomposition, and (2) perform dynamic programming along the decomposition. The goal is to have a fast runtime, and this is usually expressed as a function of some known graph width parameter of the incidence graph $I(F)$ of the formula F, like its tree-width. Step (1) is solved by a known graph algorithm for computing a decomposition of low (tree-)width, while step (2) solves #SAT or MaxSAT by dynamic programming with runtime expressed in terms of the (tree-)width k of the decomposition.

The algorithms we give in this paper also work by dynamic programming along a decomposition, but in a slightly different framework. Since we are not solving a graph theoretic problem, expressing runtime by a graph theoretic parameter may be a limitation. Therefore, our strategy will be to develop a framework based on the following strategy

(A) consider, for #SAT or MaxSAT, the amount of information needed to combine solutions to subproblems into global solutions, then
(B) define the notion of good decompositions based on a parameter that minimizes this information, and then
(C) design a dynamic programming algorithm along such a decomposition with runtime expressed by this parameter

Both Paulusma et al [25] and Slivovsky and Szeider [33] consider two assignments to be equivalent if they satisfy the same set of clauses. When carrying out (A) for #SAT and MaxSAT this led us to the concept of `ps`-value of a CNF formula. Let us define it and give an intuitive explanation. A subset $\mathcal{C}$ of the clauses of a CNF formula F is called *projection satisfiable* if there is some complete assignment satisfying every clause in $\mathcal{C}$ but not satisfying any clause not in $\mathcal{C}$. The `ps`-value of F is the number of projection satisfiable subsets of clauses. Let us consider its connection to dynamic programming, which in general applies when an optimal solution can be found by combining optimal solutions to certain

[1] $I(F)$ is the bipartite incidence graph between the clauses of F on the one hand and the variables of F on the other hand. Information about positive or negative occurrences of variables is not encoded in $I(F)$ so sometimes a signed or directed version is used that includes also this information.

subproblems. For #SAT and MaxSAT these subproblems, at least in the cases we consider, take the form of a subformula of F induced by a subset S of clauses and variables, i.e. first remove from F all variables not in S and then remove all clauses not in S. Consider for simplicity the two subproblems F_S and $F_{\overline{S}}$ defined by S and its complement $\overline{S}$. When combining the 'solutions' to F_S and $F_{\overline{S}}$, in order to find solutions to F, it seems clear that we must consider a number of cases at least as big as the ps-values of the two disjoint subformulas 'crossing' between S and $\overline{S}$, i.e. the subformulas obtained by removing from clauses in S the variables of S, and by removing from clauses in $\overline{S}$ the variables of $\overline{S}$. See Figure 2 for an example.

We did not find in the literature a study of the ps-value of CNF formulas, so we start by asking for a characterization of formulas having low ps-value. We were led to the concept of the mim-value of $I(F)$, which is the size of a maximum induced matching of $I(F)$, where an induced matching is a subset M of edges with the property that any edge of the graph is incident to at most one edge in M. Note that this value can be much lower than the size of a maximum matching, e.g. any complete bipartite graph has mim-value 1. We show that the ps-value of F is upper bounded by the number of clauses of F raised to the power of the mim-value of $I(F)$, plus 1. For a CNF formula F where $I(F)$ has mim-value 1 the interpretation of this result is straightforward: its clauses can be totally ordered such that for any two clauses $C < C'$ the variables occurring in C are a subset of the variables occurring in C', and this has the implication that the number of subsets of clauses for which some complete assignment satisfies exactly this subset is at most the number of clauses plus 1.

Families of CNF formulas having small ps-value are themselves of algorithmic interest, but in this paper we continue with part (B) of the above strategy, and focus on how to decompose a CNF formula F based on the concept of ps-value. A common way to decompose a mathematical object is to recursively partition its ground set into two parts, giving a binary tree whose root represents the ground set and whose leaves are bijectively mapped to the elements of the ground set. Taking the ground set of F to be the set containing its clauses and its variables, this is how we will decompose F, in other words by a binary tree whose leaves are in 1-1 correspondence with the variables and clauses. A node of the binary tree represents the subset X of variables and clauses at the leaves of its subtree. Which decomposition trees are good for efficiently solving #SAT and MaxSAT? In accordance with the above discussion under part (A) the answer is that the good decomposition trees are those where all subformulas 'crossing' between X and $\overline{X}$, for some X defined by a node of the tree, have low ps-value. See Figure 2 for an example. To define this informal notion precisely we use the concept of a branch decomposition over the ground set of a formula with cut function being the ps-value of the formulas crossing the cut. Branch decompositions are by now a standard notion in graph and matroid theory, originating in the work of Robertson and Seymour on graph minors [29]. This way we arrive at the definition of the ps-width of a CNF formula F, and of the decompositions of F that achieve this ps-width. It is important to note that a formula can have ps-value exponential in formula size while ps-width is polynomial, and that in general the class of formulas of low ps-width is much larger than the class of formulas of low ps-value.

To finish the above strategy, we must carry out part (C) and show how to solve #SAT and MaxSAT by dynamic programming along the branch decomposition of the formula, and express its runtime as a function of the ps-width. This is not complicated, as dynamic programming when everything has been defined properly simply becomes an exercise in brute-force computation of the sufficient and necessary information, but it is technical and quite tedious. It leads to the following theorem.

Theorem 4 *Given a formula F over n variables and m clauses, and a decomposition of F of* ps*-width k, we solve* #SAT *and weighted* MaxSAT *in time $\mathcal{O}(k^3 m(m+n))$.*

Thus, given a decomposition having a ps-width k that is *polynomially-bounded* in the number of variables n and clauses m of the formula, we get polynomial-time algorithms. Let us compare our result to the strongest previous result in this direction, namely that of Slivovsky and Szeider [33] for #SAT. Their algorithm takes as input a branch decomposition over the vertex set of $I(F)$, which is the same as the ground set of F, and evaluates its runtime by the cut function they call 'index'. They show that this cut function is closely related to the symmetric clique-width scw of the given decomposition, giving runtime $(n+m)^{\mathcal{O}(scw)}$. Considering the clique-width cw of the given decomposition the runtime of [33] becomes $(n+m)^{\mathcal{O}(2^{cw})}$ since symmetric clique-width and clique-width is related by the essentially tight inequalities $0.5cw \leq scw \leq 2^{cw}$ [12]. Their algorithm is thus a polynomial-time algorithm if given a decomposition with *constantly bounded scw*. The result of Theorem 4 encompasses this, since our Corollary 6 ties ps-width to mim-width and Vatshelle [37] shows that mim-width is upper bounded by clique-width, see also [28] for symmetric clique-width, so that a decomposition of $I(F)$ having constantly bounded (symmetric) clique-width also has polynomially bounded ps-width. In this way, given the decomposition assumed as input in [33], the algorithm of Theorem 4 will have runtime $\mathcal{O}(m^{3cw}s)$, for cw the clique-width of the given decomposition.

In a paper by Brault-Baron et al [9], appearing after a preliminary presentation of our results [31], it is argued that the framework behind Theorem 4 gives a uniform explanation of all tractability results for #SAT in the literature, in particular those using dynamic programming based on structural decompositions of the incidence graph. Brault-Baron et al [9] also goes beyond this, giving a polynomial-time algorithm, *not* by dynamic programming, to solve #SAT on β-acyclic CNF formulas, being exactly those formulas whose incidence graphs are chordal bipartite. They show that these formulas do *not* have bounded ps-width and that their incidence graphs do *not* have bounded mim-width. See Figure 1 which gives an overview of the results in this paper and in other papers.

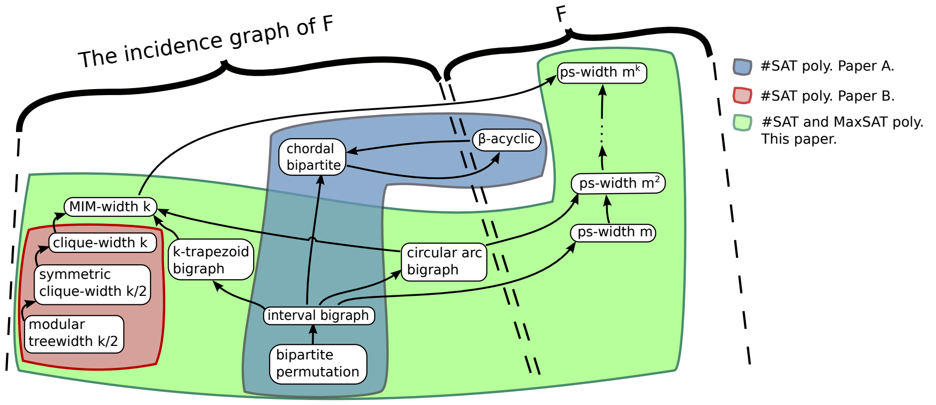


Fig. 1. We believe, as argued in [9], that any dynamic programming approach working along a structural decomposition to solve #SAT (or MaxSAT) in polynomial time cannot go beyond the green box. Paper A is [9] and Paper B is [33]. On the left of the two dashed lines are 4 classes of graphs with bound $k/2$ or k on some structural graph width parameter, and 5 classes of bipartite graphs. On the right are β-acyclic CNF formulas and 3 classes of CNF formulas with ps-width varying from linear in the number of clauses m, to m^2 and m^k. There is an arc from P to Q if any formula F or incidence graph $I(F)$ having property P also has property Q. This is a Hasse diagram, so lack of an arc in the transitive closure means this relation provably does not hold.

Using the concept of `mim`-width of graphs, introduced in the thesis of Vatshelle [37], and the connection between `ps`-value and `mim`-value alluded to earlier, we show that a rich class of formulas, including classes of unbounded clique-width, have polynomially bounded `ps`-width and are thus covered by Theorem 4. Firstly, this holds for classes of formulas having incidence graphs that can be represented as intersection graphs of certain objects, like interval graphs [4]. Secondly, it holds also for the much larger class of bipartite graphs achieved by taking bigraph bipartizations of these intersection graphs, obtained by imposing a bipartition on the vertex set and keeping only edges between the partition classes. Some such bigraph bipartizations have been studied previously, in particular the interval bigraphs. The interval bigraphs contain all bipartite permutation graphs, and these latter graphs have been shown to have unbounded clique-width [8]. See Figure 1.

Let us discuss step (1), finding a good decomposition. Note that Theorem 4 assumes that the input formula is given along with a decomposition of some `ps`-width k. The value k need not be optimal, so any heuristic finding a reasonable branch decomposition could be used in practice. Computing decompositions of optimal `ps`-width is probably not doable in polynomial-time, but the complexity of this question is not adressed in this paper. However, we are able to efficiently decide if a CNF formula has a certain linear structure guaranteeing low `ps`-width. By combining an alternative definition of interval bigraphs [18] with a fast recognition algorithm [24,26] we arrive at the following. Say that a CNF formula

F has an interval ordering if there exists a total ordering of variables and clauses such that for any variable x occurring in clause C, if x appears before C then any variable between them also occurs in C, and if C appears before x then x occurs also in any clause between them.

Theorem 11 *F over n variables and m clauses each of at most t literals. In time $\mathcal{O}((m+n)mn)$ we can decide if F has an interval ordering (yes iff $I(F)$ is an interval bigraph), and if yes we solve* #SAT *and weighted* MaxSAT *with an additional runtime of $\mathcal{O}(\min\{m^2, 4^t\}(m+n)m)$.*

Formulas with an interval ordering are precisely those whose incidence graphs are interval bigraphs, so Theorem 11 encompasses classes of formulas whose incidence graphs have unbounded clique-width.

Could parts of our algorithms be of interest for practical applications? Answering this question is beyond the scope of the present paper. However, we have performed some limited testing, in particular for formulas with a linear structure, as a simple proof of concept. All our code can be found online [1]. We have designed and implemented a heuristic for step (1) finding a good decomposition, in this case a linear one where the binary tree describing the decomposition is a path with attached leaves. We have also implemented step (2) dynamic programming solving #SAT and MaxSAT along such decompositions. We then run (1) followed by (2) and compare against one of the best MaxSAT solvers from the Max-SAT-2014 event of the SAT-2014 conference and the latest version of the #SAT solver called sharpSAT developed by Marc Thurley [36]. These solvers beat our implementation on most inputs, which is not suprising since our code does not include any techniques beyond our algorithm. Nevertheless, we were able to generate some classes of CNF formulas having interval orderings where our implementation is by far the better. This lends support to our belief that methods related to `ps`-value warrants further research to investigate if they could be useful in practice.

Our paper is organized as follows. In Section 2 we give formal definitions of `ps`-value and `ps`-width of a CNF formula and show the central combinatorial lemma linking `ps`-value of a formula to the size of the maximum induced matching in the incidence graph of the formula. In Section 3 we present dynamic programming algorithms that given a formula and a decomposition solves #SAT and weighted MaxSAT, proving Theorem 4. In Section 4 we investigate classes of formulas having decompositions of low `ps`-width, basically proving the correctness of the hierarchy presented in Figure 1. In Section 5 we consider formulas having an interval ordering and prove Theorem 11. In Section 6 we present the results of the implementations and testing. We end in Section 7 with some open problems.

2 Framework

We consider propositional formulas in Conjunctive Normal Form (CNF). A *literal* is a propositional *variable* or a negated variable, x or $\neg x$, a *clause* is a set of literals, and a *formula* is a multiset of clauses. For a formula F, $\texttt{cla}(F)$ denotes the clauses in F. The incidence graph of a formula F is the bipartite graph $I(F)$ having a vertex for each clause and variable, with variable x adjacent to any clause C in which it occurs. We consider only input formulas where $I(F)$ is connected, as otherwise we would solve our problems on the separate components of $I(F)$. For a clause C, $\texttt{lit}(C)$ denotes the set of literals in C and $\texttt{var}(C)$ denotes the variables of the literals in $\texttt{lit}(C)$. For a formula F, $\texttt{var}(F)$ denotes the union $\bigcup_{C \in \texttt{cla}(F)} \texttt{var}(C)$. For a set X of variables, an *assignment* of X is a function $\tau : X \to \{0, 1\}$. For a literal ℓ, we define $\tau(\ell)$ to be $1 - \tau(\texttt{var}(\ell))$ if ℓ is a negated variable ($\ell = \neg x$ for some variable x) and to be $\tau(\texttt{var})$ otherwise ($\ell = x$ for some variable x). A clause C is said to be *satisfied* by an assignment τ if there exists at least one literal $\ell \in \texttt{lit}(C)$ so that $\tau(\ell) = 1$. All clauses an assignment τ do not satisfy are said to be *falsified* by τ. We notice that this means an empty clause will be falsified by all assignments. A formula is satisfied by an assignment τ if τ satisfies all clauses in $\texttt{cla}(F)$.

The problem #SAT, given a formula F, asks how many distinct assignments of $\texttt{var}(F)$ satisfy F. The optimization problem weighted MaxSAT, given a formula F and weight function $w : \texttt{cla}(F) \to \mathbb{N}$, asks what assignment τ of $\texttt{var}(F)$ maximizes $\sum_C w(C)$ for all $C \in \texttt{cla}(F)$ satisfied by τ. The problem MaxSAT is weighted MaxSAT where all clauses have weight one. For weighted MaxSAT, we assume the sum of all the weights are at most $2^{O(\texttt{cla}(F))}$, and thus we can do summation on the weights in time linear in $\texttt{cla}(F)$.

For a set A, with elements from a universe U we denote by $\overline{A}$ the elements in $U \setminus A$, as the universe is usually given by the context.

2.1 Cut of a formula

In this paper, we will solve MaxSAT and #SAT by the use of dynamic programming. We will be using a divide and conquer technique where we solve the problem on smaller subformulas of the original formula F and then combine the solutions to each of these smaller formulas to form a solution to the entire formula F. Note however, that the solutions found for a subformula will depend on the interaction between the subformula and the remainder of the formula. We use the following notation for subformulas.

For a clause C and set X of variables, by $C|_X$ we denote the clause $\{\ell \in C : \texttt{var}(\ell) \in X\}$. We say $C|_X$ is the clause C *induced* by X. Unless otherwise specified, all clauses mentioned in this paper is from the set $\texttt{cla}(F)$ (e.g., if we write $C|_X \in \texttt{cla}(F')$, we still assume $C \in \texttt{cla}(F)$). For a formula F and subsets $\mathcal{C} \subseteq \texttt{cla}(F)$ and $X \subseteq \texttt{var}(F)$, we say the subformula $F_{\mathcal{C},X}$ of F *induced* by $\mathcal{C}$ and X is the formula consisting of the clauses $\{C_i|_X : C_i \in \mathcal{C}\}$. That is, $F_{\mathcal{C},X}$ is the formula we get by removing all clauses not in $\mathcal{C}$ followed by removing each literal of a variable not in X. For a set $\mathcal{C}$ of clauses, we denote by $\mathcal{C}|_X$ the set $\{C|_X : C \in \mathcal{C}\}$. As with a clause, for an assignment τ over a set X of variables, we say the assignment τ *induced* by $X' \subseteq X$ is the assignment $\tau|_{X'}$ where the domain is restricted to X'.

For a formula F and sets $\mathcal{C} \subseteq \texttt{cla}(F)$, $X \subseteq \texttt{var}(F)$, and $S = \mathcal{C} \cup X$, we call S a *cut* of F and note that it breaks F into four subformulas $F_{\mathcal{C},X}$, $F_{\overline{\mathcal{C}},X}$, $F_{\mathcal{C},\overline{X}}$, and $F_{\overline{\mathcal{C}},\overline{X}}$. See Figure 2. One important fact we may observe from this definition is that a clause C in F is satisfied by an assignment τ of $\mathtt{var}(F)$, if and only if C (induced by X or $\overline{X}$) is satisfied by τ in at least one of the formulas of any cut of F.

2.2 Projection satisfiable sets and ps-value of a formula

For a formula F and assignment τ of some of the variables in $\mathtt{var}(F)$, we denote by $\mathtt{sat}(F,\tau)$ the inclusion maximal set $\mathcal{C} \subseteq \mathtt{cla}(F)$ so that each clause in $\mathcal{C}$ is satisfied by τ. If for a set $\mathcal{C} \subseteq \mathtt{cla}(F)$ we have $\mathtt{sat}(F,\tau) = \mathcal{C}$ for some τ over all the variables in $\mathtt{var}(F)$, then $\mathcal{C}$ is known as a *projection* (see e.g. [21,33]) and we say $\mathcal{C}$ is *projection satisfiable* in F. We denote by $\mathtt{PS}(F)$ the family of all projection satisfiable sets in F. That is,

$$\mathtt{PS}(F) = \{\mathtt{sat}(F,\tau) : \tau \text{ is an assignment of the entire set } \mathtt{var}(F)\}.$$

The cardinality of this set, $|\mathtt{PS}(F)|$, is referred to as the ps-value of F.

To get a grasp of the structure of formulas having low ps-value we consider induced matchings in the incidence graph of a formula. The incidence graph of a formula F is the bipartite graph $I(F)$ having a vertex for each clause and variable, with variable x adjacent to any clause C in which it occurs. An induced matching in a graph is a subset M of edges with the property that any edge of the graph is incident to at most one edge in M. In other words, for any 3 vertices a,b,c, if ab is an edge in M and bc is an edge then there does not exist an edge cd in M. The number of edges in M is called the size of the induced matching. The following result provides an upper bound on the ps-value of a formula in terms of the maximum size of an induced matching of its incidence graph.

Lemma 1. *Let F be a CNF formula with no clause containing more than t literals, and let k be the maximum size of an induced matching in $I(F)$. We then have $|\mathtt{PS}(F)| \leq \min\{|\mathtt{cla}(F)|^k + 1, 2^{tk}\}$.*

Proof. We first argue that $|\mathtt{PS}(F)| \leq |\mathtt{cla}(F)|^k + 1$. Let $\mathcal{C} \in \mathtt{PS}(F)$ and $\mathcal{C}_f = \mathtt{cla}(F) \setminus \mathcal{C}$. Thus, there exists a complete assignment τ such that the clauses not satisfied by τ are $\mathcal{C}_f = \mathtt{cla}(F) \setminus \mathtt{sat}(F,\tau)$. Since every variable in $\mathtt{var}(F)$ appears in some clause of F this means that $\tau|_{\mathtt{var}(\mathcal{C}_f)}$ is the unique assignment of the variables in $\mathtt{var}(\mathcal{C}_f)$ which do not satisfy any clause of $\mathcal{C}_f$. Let $\mathcal{C}_f^{'} \subseteq \mathcal{C}_f$ be an inclusion minimal set such that $\mathtt{var}(\mathcal{C}_f) = \mathtt{var}(\mathcal{C}_f^{'})$, hence $\tau|_{\mathtt{var}(\mathcal{C}_f)}$ is also the unique assignment of the variables in $\mathtt{var}(\mathcal{C}_f)$ which do not satisfy any clause of $\mathcal{C}_f^{'}$. An upper bound on the number of different such minimal $\mathcal{C}_f^{'}$, over all $\mathcal{C} \in \mathtt{PS}(F)$, will give an upper bound on $|\mathtt{PS}(F)|$. For every $C \in \mathcal{C}_f^{'}$ there is a variable v_C appearing in C and no other clause of $\mathcal{C}_f^{'}$, otherwise $\mathcal{C}_f^{'}$ would not be minimal. Note that we have an induced matching M of $I(F)$ containing all such edges v_C, C. By assumption, the induced matching M can have at most k edges and hence $|\mathcal{C}_f^{'}| \leq k$. It is easy to show by induction on k that there are at most $|\mathtt{cla}(F)|^k + 1$ sets of at most k clauses and the lemma follows.

We now argue that $|\texttt{PS}(F)| \leq 2^{tk}$. As the maximum induced matching has size k there is some set $\mathcal{C}$ of k clauses so that $var(\mathcal{C}) = var(F)$. As each clause $C \in \mathcal{C}$ has $|var(C)| \leq t$, we have $|var(F)| = |var(\mathcal{C})| \leq tk$. As there are no more than $2^{|var(F)|}$ assignments for F, the PS-value of F is upper bounded by 2^{tk}. $\square$

2.3 The ps-width of a formula

We define a *branch decomposition* of a formula F to be a pair (T, δ) where T is a rooted binary tree and δ is a bijective function from the leaves of T to the clauses and variables of F. If all the non-leaf nodes (also referred to as *internal* nodes) of T induce a path, we say that (T, δ) is a *linear* branch decomposition. For a non-leaf node v of T, we denote by $\delta(v)$ the set $\{\delta(l) : l \text{ is a leaf in the subtree rooted in } v\}$. Based on this, we say that the decomposition (T, δ) of formula F induces certain cuts of F, namely the cuts defined by $\delta(v)$ for each node v in T.

For a formula F and branch decomposition (T, δ), for each node v in T, by F_v we denote the formula induced by the clauses in $\texttt{cla}(F) \setminus \delta(v)$ and the variables in $\delta(v)$, and by $F_{\overline{v}}$ we denote the formula on the complement sets; i.e. the clauses in $\delta(v)$ and the variables in $\texttt{var}(F) \setminus \delta(v)$. In other words, if $\delta(v) = \mathcal{C} \cup X$ with $\mathcal{C} \subseteq \texttt{cla}(F)$ and $X \subseteq \texttt{var}(F)$ then $F_v = F_{\overline{\mathcal{C}},X}$ and $F_{\overline{v}} = F_{\mathcal{C},\overline{X}}$. To simplify the notation, we will for a node v in a branch decomposition and a set $\mathcal{C}$ of clauses denote by $\mathcal{C}|_v$ the set $\mathcal{C}|_{\texttt{var}(F_v)}$. We define the $\texttt{ps}$-*value* of the cut $\delta(v)$ to be

$$\texttt{ps}(\delta(v)) = \max\{|PS(F_v)|, |PS(F_{\overline{v}})|\}$$

We define the $\texttt{ps}$-*width* of a branch decomposition to be

$$\texttt{psw}(T, \delta) = \max\{\texttt{ps}(\delta(v)) : v \text{ is a node of } T\}$$

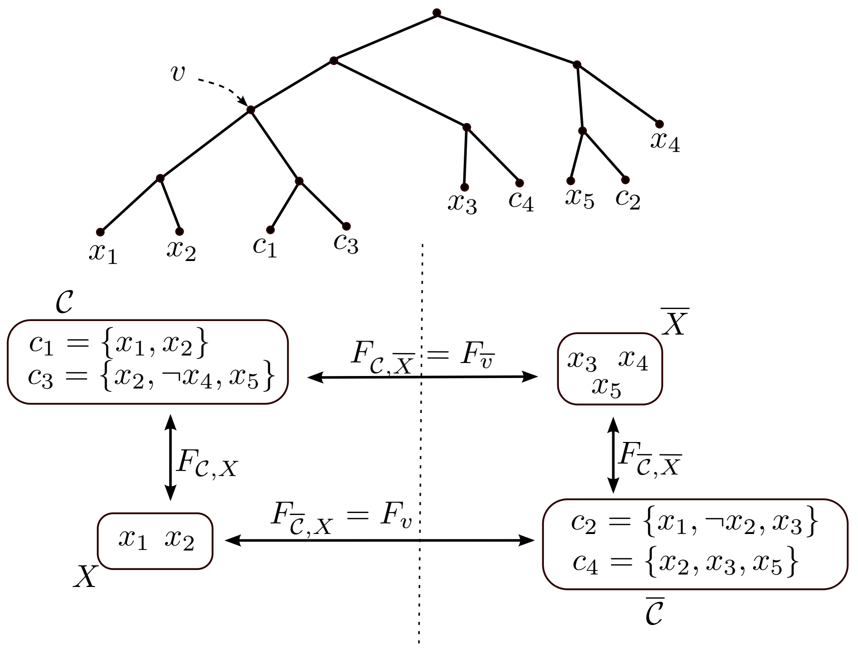
We define the $\texttt{ps}$-*width* of a formula F to be

$$\texttt{psw}(F) = \min\{\texttt{psw}(T, \delta) : (T, \delta) \text{ is a branch decomposition of } F\}$$

Note that the $\texttt{ps}$-value of a cut is a symmetric function. That is, the $\texttt{ps}$-value of cut S equals the $\texttt{ps}$-value of the cut $\overline{S}$. See Figure 2 for an example.

3 Dynamic programming for MaxSAT and #SAT

Given a branch decomposition (T, δ) of a CNF formula F over n variables and m clauses and of total size s, we will give algorithms that solve MaxSAT and #SAT on F in time $\mathcal{O}(\texttt{psw}(T, \delta)^3 m(m+n))$. Our algorithms are strongly inspired by the algorithm of [33], but in order to achieve a runtime polynomial in $\texttt{ps}$-width, and also to solve MAXSAT, we must make some crucial changes. In particular, we must index the dynamic programming tables by PS-sets rather than the 'shapes' used in [33].



v
x_1
x_2
c_1
c_3
x_3
c_4
x_5
c_2
x_4
$\mathcal{C}$
$c_1 = \{x_1, x_2\}$
$c_3 = \{x_2, \neg x_4, x_5\}$
$F_{\mathcal{C},\overline{X}} = F_{\overline{v}}$
$\overline{X}$
x_3 x_4 x_5
$F_{\mathcal{C},X}$
$F_{\overline{\mathcal{C}},\overline{X}}$
$F_{\overline{\mathcal{C}},X} = F_v$
x_1 x_2
X
$c_2 = \{x_1, \neg x_2, x_3\}$
$c_4 = \{x_2, x_3, x_5\}$
$\overline{\mathcal{C}}$

Fig. 2. On top is a branch decomposition of a formula F with $\texttt{var}(F) = \{x_1, x_2, x_3, x_4, x_5\}$ and the 4 clauses $\texttt{cla}(F) = \{c_1, c_2, c_3, c_4\}$ as given in the boxes. The node v of the tree defines the cut $\delta(v) = \mathcal{C} \cup X$ where $\mathcal{C} = \{c_1, c_3\}$ and $X = \{x_1, x_2\}$. There are 4 subformulas defined by this cut: $F_{\mathcal{C},X}, F_{\overline{\mathcal{C}},\overline{X}}, F_{\overline{\mathcal{C}},X}, F_{\mathcal{C},\overline{X}}$. For example, $F_{\overline{\mathcal{C}},X} = \{\{x_1, \neg x_2\}, \{x_2\}\}$ and $F_{\mathcal{C},\overline{X}} = \{\emptyset, \{\neg x_4, x_5\}\}$. We have $F_v = F_{\overline{\mathcal{C}},X}$ and $F_{\overline{v}} = F_{\mathcal{C},\overline{X}}$ with projection satisfiable sets of clauses $\texttt{PS}(F_v) = \{\{c_2|_v\}, \{c_4|_v\}, \{c_2|_v, c_4|_v\}\}$ and $\texttt{PS}(F_{\overline{v}}) = \{\emptyset, \{c_3|_{\overline{v}}\}\}$ and the $\texttt{ps}$-value of this cut is $\texttt{ps}(\delta(v)) = \max\{|PS(F_v)|, |PS(F_{\overline{v}})|\} = 3$.

Special terminology. In this dynamic programming section, we will combine partial solutions to subformulas into solutions for the input formula F. To improve readability we introduce notation PS' and $\texttt{sat}'$ that allows us to refer directly to the clauses of F, also when working on the subformulas. Thus, for a formula F and branch decomposition (T, δ), for each node v in T, and induced subformula F_v of F, by $\texttt{PS}'(F_v)$ we denote the subsets of clauses $\mathcal{C}$ from $\texttt{cla}(F) \setminus \delta(v)$ so that $\texttt{PS}(F_v) = \mathcal{C}|_{\texttt{var}(F_v)}$. Similarly, for an assignment τ over $\texttt{var}(F_v)$, by $\texttt{sat}'(F_v, \tau)$ we denote the set of clauses $\mathcal{C}$ from $\texttt{cla}(F) \setminus \delta(v)$ so that $\texttt{sat}(F_v, \tau) = \mathcal{C}|_{\texttt{var}(F_v)}$. Note that $|\texttt{PS}'(F_v)| = |\texttt{PS}(F_v)|$ and $|\texttt{sat}'(F_v, \tau)| = |\texttt{sat}(F_v, \tau)|$. We take the liberty to call also these sets projection satisfiable and refer to them as 'PS-sets' in the text, but it will be clear from context that we mean clauses of $\texttt{cla}(F)$ and not $\texttt{cla}(F_v)$.

Implementation details. We regard PS-sets as boolean vectors of length $|\texttt{cla}(F)|$, and assume we can identify clauses and variables by integer numbers. So, checking if a clause is in a PS-set can be done in constant time, and checking if two PS-sets are equal can be done in $\mathcal{O}(|\mathtt{cla}(F)|)$ time. To manage our PS-sets, we use a binary `trie` datastructure (see [15]). We can add and retrieve a PS-set to and from a `trie` in $\mathcal{O}(|\mathtt{cla}(F)|)$ time. Trying to add a PS-set to a `trie` already containing an equivalent PS-set will not alter the content of the `trie`, so our `trie`'s will only contain distinct PS-sets. As retrieval of an element in our `trie` takes $\mathcal{O}(|\mathtt{cla}(F)|)$ time, by assigning a distinct integer to each PS-set at the time it is added to the `trie`, we have a $\mathcal{O}(|\mathtt{cla}(F)|)$-time mapping from PS-sets to distinct integers. This will be used implicitly in our algorithms when we say we index by PS-sets; when implementing the algorithm we will instead index by the according distinct integer the PS-set is mapped to.

In a pre-processing step we will need the following which, for each node v in T computes the sets of projection satisfiable subsets of clauses $\mathtt{PS}'(F_v)$ and $\mathtt{PS}'(F_{\overline{v}})$ of the two crossing subformulas F_v and $F_{\overline{v}}$.

Theorem 2. *Given a CNF formula F with a branch decomposition (T, δ) of* $\mathtt{ps}$*-width k, we can in time $\mathcal{O}(k^2 m(m+n))$ compute the sets $\mathtt{PS}'(F_v)$ and $\mathtt{PS}'(F_{\overline{v}})$ for each v in T.*

Proof. We notice that for a node v in T with children c_1 and c_2, we can express $\mathtt{PS}'(F_v)$ as

$$\mathtt{PS}'(F_v) = \left\{ (C_1 \cup C_2) \cap \mathtt{cla}(F_v) : \begin{array}{l} C_1 \in \mathtt{PS}'(F_{c_1}), \text{ and} \\ C_2 \in \mathtt{PS}'(F_{c_2}) \end{array} \right\} .$$

Similarly, for sibling s and parent p of v in T, the set $\mathtt{PS}'(F_{\overline{v}})$ can be expressed as

$$\mathtt{PS}'(F_{\overline{v}}) = \left\{ (C_p \cup C_s) \cap \mathtt{cla}(F_{\overline{v}}) : \begin{array}{l} C_p \in \mathtt{PS}'(F_{\overline{p}}), \text{ and} \\ C_s \in \mathtt{PS}'(F_s) \end{array} \right\} .$$

By transforming these recursive expressions into a dynamic programming algorithm, as done in Procedure 1 and Procedure 2 below, we are able to calculate all the desired sets as long as we can compute the sets for the base cases $\mathtt{PS}'(F_l)$ when l is a leaf of T, and $\mathtt{PS}'(F_{\overline{r}})$ for the root r of T. However, these formulas contain at most one variable, and thus we can easily construct their set of projection satisfiable clauses in linear amount of time for each of the formulas. For the rest of the formulas, we construct the formulas using Procedure 1 and Procedure 2. As there are at most twice as many nodes in T as there are clauses and variables in F, the procedures will run at most $\mathcal{O}(|\mathtt{cla}(F)| + |\mathtt{var}(F)|)$ times. In each run of the algorithms, we iterate through at most k^2 pairs of projection satisfiable sets, and do a constant number of set operations that might take $\mathcal{O}(|\mathtt{cla}(F)|)$ time each. This results in a total runtime of $\mathcal{O}(k^2|\mathtt{cla}(F)|(|\mathtt{cla}(F)| + |\mathtt{var}(F)|)) = \mathcal{O}(k^2 m(m+n))$ for all the nodes of T combined. □

We now move on to the dynamic programming proper. We first give the algorithm for MaxSAT and then briefly describe the changes necessary for solving weighted MaxSAT and #SAT.

```
Procedure 1: Generating $\mathtt{PS}'(F_v)$
input:  $\mathtt{PS}'(F_{c_1})$ and $\mathtt{PS}'(F_{c_2})$ for children $c_1$ and $c_2$ of $v$
        in branch decomposition
output: $\mathtt{PS}'(F_v)$

$L \leftarrow$ empty trie of projection satisfiable clause-sets
for each $(C_1, C_2) \in \mathtt{PS}'(F_{c_1}) \times \mathtt{PS}'(F_{c_2})$ do
   add $(C_1 \cup C_2) \cap \mathtt{cla}(F_v)$ to $L$
return $L$
```

```
Procedure 2: Generating $\mathtt{PS}'(F_{\overline{v}})$
input:  $\mathtt{PS}'(F_s)$ and $\mathtt{PS}'(F_{\overline{p}})$ for sibling $s$ and parent $p$ of $v$
        in branch decomposition
output: $\mathtt{PS}'(F_{\overline{v}})$

$L \leftarrow$ empty trie of projection satisfiable clause-sets
for each $(C_s, C_p) \in \mathtt{PS}'(F_s) \times \mathtt{PS}'(F_{\overline{p}})$ do
   add $(C_s \cup C_p) \cap \mathtt{cla}(F_{\overline{v}})$ to $L$
return $L$
```

Our algorithm uses the technique of 'expectation' introduced in [10,11]. Some partial solutions might be good when combined with certain partial solutions, but bad when combined with others. In the technique of 'expectation' we categorize how partial solutions can interact, and then optimize our selection of partial solutions based on the 'expectation' that this interaction occurs. In our dynamic programming algorithm for MaxSAT, we apply this technique by making expectations on each cut regarding what set of clauses will be satisfied by variables of the opposide side of the cut.

For a node v in the decomposition of F and PS-sets $C \in \mathtt{PS}'(F_v)$ and $C' \in \mathtt{PS}'(F_{\overline{v}})$, we say that an assignment τ of $\mathtt{var}(F)$ *meets the expectation C and C'* if $\mathtt{sat}'(F_v, \tau|_v) = C$ and $\mathtt{sat}'(F_{\overline{v}}, \tau|_{\overline{v}}) = C'$. For each node v of the branch decomposition, our algorithm uses a table $\mathtt{Tab}_v$ that for each pair $(C, C') \in \mathtt{PS}'(F_v) \times \mathtt{PS}'(F_{\overline{v}})$ stores in $\mathtt{Tab}_v(C, C')$ the maximum number of clauses in $\delta(v)$ that are satisfied, over all assignments meeting the expectation of C and C'. As the variables in $\mathtt{var}(F) \setminus \delta(v)$ satisfy exactly C', for any assignment that meets this expectation, an equivalent formulation of the content of $\mathtt{Tab}_v(C, C')$ is that it must satisfy the following constraint:

$$\begin{gathered}\text{Over all assignments } \tau \text{ of } \mathtt{var}(F) \cap \delta(v) \text{ such that } \mathtt{sat}'(F_v, \tau) = C \;, \\ \mathtt{Tab}_v(C, C') = \max_{\tau} \{\, |\, (\mathtt{sat}'(F, \tau') \cap \delta(v)) \cup C' \,|\, \}\end{gathered} \tag{1}$$

By bottom-up dynamic programming along the tree T we compute the tables of each node of T. For a leaf l in T, generating $\mathtt{Tab}_l$ can be done easily in linear time since the formula F_v contains at most one variable. For an internal node v of T, with children c_1, c_2, we compute $\mathtt{Tab_v}$ by the algorithm described in Procedure 3. There are 3 tables involved in this update, one at each child and one at the parent. A pair of entries, one from each child table, may lead to an update of an entry in the parent table. Each table entry is indexed by a pair, thus there are 6 indices involved in a single potential update. A trick first introduced in [11] allows us to loop over triples of indices and for each triple compute the remaining 3 indices forming the 6-tuple involved in the update, thereby reducing the runtime.

```
Procedure 3: Computing $\mathtt{Tab_v}$ for inner node $v$ with children $c_1, c_2$
input:  $\mathtt{Tab_{c_1}}$, $\mathtt{Tab_{c_2}}$
output: $\mathtt{Tab_v}$
1. initialize $\mathtt{Tab_v} : \mathtt{PS}'(F_v) \times \mathtt{PS}'(F_{\overline{v}}) \to \{-1\}$
2. for each $(C_{c_1}, C_{c_2}, C'_v)$ in $\mathtt{PS}'(F_{c_1}) \times \mathtt{PS}'(F_{c_2}) \times \mathtt{PS}'(F_{\overline{v}})$ do
3.     $C'_{c_1} \leftarrow (C_{c_2} \cup C'_v) \cap \delta(c_1)$
4.     $C'_{c_2} \leftarrow (C_{c_1} \cup C'_v) \cap \delta(c_2)$
5.     $C_v \leftarrow (C_{c_1} \cup C_{c_2}) \setminus \delta(v)$
6.     $t \leftarrow \mathtt{Tab_{c_1}}(C_{c_1}, C'_{c_1}) + \mathtt{Tab_{c_2}}(C_{c_2}, C'_{c_2})$
7.     if $\mathtt{Tab_v}(C_v, C'_v) < t$ then $\mathtt{Tab_v}(C_v, C'_v) \leftarrow t$
8. return $\mathtt{Tab_v}$
```

Lemma 3. *For a CNF formula F of m clauses and an inner node v, of a branch decomposition (T, δ) of* $\mathtt{ps}$*-width k, Procedure 3 computes* $\mathtt{Tab_v}$ *satisfying Constraint (1) in time $\mathcal{O}(k^3 m)$.*

Proof. We assume $\mathtt{Tab_{c_1}}$ and $\mathtt{Tab_{c_2}}$ satisfy Constraint (1). Procedure 3 loops over all triples in $\mathtt{PS}'(F_{c_1}) \times \mathtt{PS}'(F_{c_2}) \times \mathtt{PS}'(F_{\overline{v}})$. From the definition of $\mathtt{ps}$-width of (T, δ) there are at most k^3 such triples. Each operation inside an iteration of the loop take $\mathcal{O}(m)$ time and there is a constant number of such operations. Thus the runtime is $\mathcal{O}(k^3 m)$.

Before we show the correctness of the output, let us look a bit at the workings of Procedure 3. For any assignment τ over $\mathtt{var}(F)$, and cut, the assignment τ will only meet the expectation of a single pair of PS-sets. Let (X_1, X'_1), (X_2, X'_2) and (X_v, X'_v) be the pairs an assignment τ meets the expectation for with respect to the cuts induced by c_1, c_2, and v, respectively. We notice that

$$\begin{aligned}
X_v &= \mathtt{sat}'(F_v, \tau|_v) \\
&= \mathtt{sat}'(F_v, \tau|_{c_1} \uplus \tau|_{c_2}) \\
&= \mathtt{sat}'(F_v, \tau|_{c_1}) \cup \mathtt{sat}'(F_v, \tau|_{c_2}) \\
&= (\mathtt{sat}'(F_{c_1}, \tau|_{c_1}) \setminus \delta(v)) \cup (\mathtt{sat}'(F_{c_2}, \tau|_{c_2}) \setminus \delta(v)) \\
&= (X_1 \setminus \delta(v)) \cup (X_2 \setminus \delta(v)) \\
&= (X_1 \cup X_2) \setminus \delta(v).
\end{aligned} \tag{2}$$

This can also be seen from Figure 3. By symmetry, we find similar values for X'_1 and X'_2; namely $X'_1 = (X_2 \cup X'_v) \cap \delta(c_1)$ and $X'_2 = (X_1 \cup X'_v) \cap \delta(c_2)$. So, these latter three sets will be implicit based on the three former sets with respect to the cuts induced by v, c_1 and c_2. We will therefore, for convenience of this

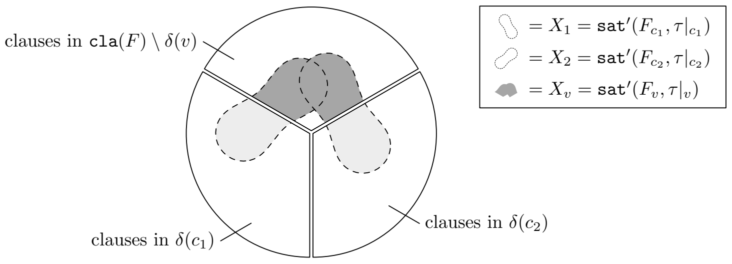

clauses in $\mathtt{cla}(F) \setminus \delta(v)$
clauses in $\delta(c_1)$
clauses in $\delta(c_2)$
$= X_1 = \mathtt{sat}'(F_{c_1}, \tau|_{c_1})$
$= X_2 = \mathtt{sat}'(F_{c_2}, \tau|_{c_2})$
$= X_v = \mathtt{sat}'(F_v, \tau|_v)$

Fig. 3. As shown by the chain of equalities in (2) in the proof of Lemma 3, the clauses in $\mathtt{sat}'(F_v, \tau|_v)$ are precisely the clauses in $(\mathtt{sat}'(F_{c_1}, \tau|_{c_1}) \cup \mathtt{sat}'(F_{c_2}, \tau|_{c_2})) \setminus \delta(v)$.

proof, say that an assignment τ *meets the expectation of a triple* (C_1, C_2, C') of PS-sets, when τ meets the expectation of the implicit three pairs on each of their respective cuts. We notice that for each choice of triples of PS-sets (C_{c_1}, C_{c_2}, C'_v) Procedure 3 computes the implicit three other sets and names them C'_{c_1}, C'_{c_2} and C_v accordingly.

We will now show that for all pairs $(C, C') \in \mathtt{PS}'(F_v) \times \mathtt{PS}'(F_{\overline{v}})$ the value of $\mathtt{Tab}_v(C, C')$ is correct. Let τ_0 be an assignment over $\mathtt{var}(F)$ that satisfies the maximum number of clauses, while meeting the expectation of C and C'. Thus, the value of $\mathtt{Tab}_v(C, C')$ is correct if and only if it stores exactly the number of clauses from $\delta(v)$ that τ_0 satisfies.

Let (C_1, C'_1) and (C_2, C'_2) be the pairs of PS-sets that τ_0 meet the expectation of for the cut $(\delta(c_1), \overline{\delta(c_1)})$ and $(\delta(c_2), \overline{\delta(c_2)})$, respectively. As τ_0 meets these expectations, the value of $\mathtt{Tab}_{c_1}(C_1, C'_1)$ and $\mathtt{Tab}_{c_2}(C_2, C'_2)$ must be at least as large as the number of clauses τ_0 satisfies in $\delta(c_1)$ and $\delta(c_2)$, respectively. Thus, the number of clauses τ_0 satisfies in both $\delta(c_1)$ and $\delta(c_2)$ is at most as large as the sum of these two entries. Since Procedure 3, in the iteration where $C'_v = C'$, $C_{c_1} = C_1$ and $C_{c_2} = C_2$, ensures that $\mathtt{Tab}_v(C, C')$ is at least the sum of $\mathtt{Tab}_{c_1}(C_1, C'_1)$ and $\mathtt{Tab}_{c_2}(C_2, C'_2)$, we know $\mathtt{Tab}_v(C, C')$ is at least as large as the correct value.

Now assume for contradiction that the value of the cell $\mathtt{Tab}_v(C, C')$ is too large. That means that at some iteration of Procedure 3 it is being assigned the value $\mathtt{Tab}_{c_1}(C_{c_1}, C'_{c_1}) + \mathtt{Tab}_{c_2}(C_{c_2}, C'_{c_2})$ when this sum is too large. Let τ_1 and τ_2 be the assignments of $\mathtt{var}(F)$ meeting the expectation of C_{c_1} and C'_{c_1} and meeting the expectation of C_{c_2}, C'_{c_2}, respectively, where the number of clauses of $\delta(c_1)$ and $\delta(c_2)$, respectively, equals the according table entries of $\mathtt{Tab}_{c_1}$ and $\mathtt{Tab}_{c_2}$. If we now take the assignment $\tau_x = \tau_1|_{c_1} \uplus \tau_2|_{c_2} \uplus \tau_0|_{\overline{v}}$, we have an assignment that meets the expectation of C and C', and who satisfies more clauses in $\delta(v)$ than τ_0, contradicting the choice of τ_0. So $\mathtt{Tab}_v(C, C')$ can be neither smaller nor larger than the number of clauses in $\delta(v)$ τ_0 satisfies, so it is exactly the same. $\square$

Theorem 4. *Given a formula F over n variables and m clauses, and a branch decomposition (T, δ) of F of* ps*-width k, we solve* MaxSAT*,* #SAT*, and weighted* MaxSAT *in time $\mathcal{O}(k^3 m(m+n))$.*

Proof. To solve MaxSAT, we first compute $\mathtt{Tab}_r$ for the root node r of T. This requires that we first compute $\mathtt{PS}'(F_v)$ and $\mathtt{PS}'(F_{\overline{v}})$ for all nodes v of T, and then, in a bottom up manner, compute $\mathtt{Tab}_v$ for each of the $\mathcal{O}(m+n)$ nodes in T. The former part we can do in $\mathcal{O}(k^2 m(m+n))$ time by Theorem 2, and the latter part we do in $\mathcal{O}(k^3 m(m+n))$ time by Lemma 3.

At the root r of T we have $\delta(r) = \mathtt{var}(F) \cup \mathtt{cla}(F)$. Thus $F_r = \emptyset$ and $F_{\overline{r}}$ does not have any variables, so that $PS(F_r) \times PS(F_{\overline{r}})$ contains only $(\emptyset, \emptyset)$. As all assignments over $\mathtt{var}(F)$ meet the expectation of $\emptyset$ and $\emptyset$ on the cut $(\delta(r), \overline{\delta(r)})$, and $\mathtt{cla}(F) \cap \delta(r) = \mathtt{cla}(F)$, by Constraint (1) the value of $\mathtt{Tab}_r(\emptyset, \emptyset)$ is the maximal number of clauses in F any assignment of $\mathtt{var}(F)$ satisfies. And hence, this number is the solution to MaxSAT.

For a weight function $w : \mathtt{cla}(F) \to \mathbb{N}$, by redefining Constraint (1) for $\mathtt{Tab}_v$ to maximize $w(\mathtt{sat}'(F, \tau) \cap \delta(v))$ instead of $|\mathtt{sat}'(F, \tau) \cap)|$, we are able to solve the more general problem weighted MaxSAT in the same way.

For the problem #SAT, we care only about assignments satisfying all the clauses of F, and we want to decide the number of distinct assignments doing so. This requires a few alterations. Firstly, alter the definition of the contents of $\mathtt{Tab}_v(C, C')$ in Constraint (1) to be the number of assignments τ over $\mathtt{var}(F) \cap \delta(v)$ where $\mathtt{sat}'(F_v, \tau) = C$ and all clauses in $\delta(v)$ is either in C' or satisfied by τ. Secondly, when computing $\mathtt{Tab}_l$ for the leaves l of T, we set each of the entries of $\mathtt{Tab}_l$ to either zero, one, or two, according to the definition. Thirdly, we alter the algorithm to compute $\mathtt{Tab}_v$ (Procedure 3) for inner nodes. We initialize $\mathtt{Tab_v}(C, C')$ to be zero at the start of the algorithm, and substitute lines 6 and 7 of Procedure 3 by the following line which increases the table value by the product of the table values at the children

$$\mathtt{Tab_v}(C_v, C_{\overline{v}}) \leftarrow \mathtt{Tab_v}(C_v, C_{\overline{v}}) + \mathtt{Tab_{c_1}}(C_{c_1}, C_{\overline{c_1}}) \cdot \mathtt{Tab_{c_2}}(C_{c_2}, C_{\overline{c_2}})$$

This will satisfy our new constraint of $\mathtt{Tab}_v$ for internal nodes v of T. The value of $\mathtt{Tab}_r(\emptyset, \emptyset)$ at the root r of T will be exactly the number of distinct assignments satisfying all clauses of F. □

The bottleneck giving the cubic factor k^3 in the runtime of Theorem 4 is the number triples in $\mathtt{PS}'(F_{\overline{v}}) \times \mathtt{PS}'(F_{c_1}) \times \mathtt{PS}'(F_{c_2})$ for any node v with children c_1 and c_2. When (T, δ) is a linear branch decomposition, it is always the case that either c_1 or c_2 is a leaf of T. In this case either $|\mathtt{PS}'(F_{c_1})|$ or $|\mathtt{PS}'(F_{c_2})|$ is a constant. Therefore, for linear branch decompositions $\mathtt{PS}'(F_{\overline{v}}) \times \mathtt{PS}'(F_{c_1}) \times \mathtt{PS}'(F_{c_2})$ will contain no more than $\mathcal{O}(k^2)$ triples. Thus we can reduce the runtime of the algorithm by a factor of k.

Theorem 5. *Given a formula F over n variables and m clauses, and a linear branch decomposition (T, δ) of F of* ps*-width k, we solve* #SAT*,* MaxSAT*, and weighted* MaxSAT *in time $\mathcal{O}(k^2 m(m+n))$.*

4 CNF formulas of polynomial ps-width

In this section we investigate classes of CNF formulas having decompositions with `ps`-width polynomially bounded in the total size s of the formula. In particular, we show that this holds whenever the incidence graph of the formula has constant `mim`-width (maximum induced matching-width, introduced by Vatshelle [37]). We also show that a large class of bipartite graphs, using what we call bigraph bipartizations, have constant `mim`-width.

In order to lift the upper bound of Lemma 1 on the `ps`-value of F, i.e $|\mathtt{PS}(F)|$, to the `ps`-width of F, we use `mim`-width of the incidence graph $I(F)$, which is defined using branch decompositions of graphs. A branch decomposition of the formula F, as defined in Section 2, can also be seen as a branch decomposition of the incidence graph $I(F)$. Nevertheless, for completeness, we formally define branch decompositions of graphs and `mim`-width.

A branch decomposition of a graph G is a pair (T, δ) where T is a rooted binary tree and δ a bijection between the leaf set of T and the vertex set of G. For a node w of T let the subset of $V(G)$ in bijection δ with the leaves of the subtree of T rooted at w be denoted by V_w. We say the decomposition defines the cut $(V_w, \overline{V_w})$. The `mim`-value of a cut $(V_w, \overline{V_w})$ is the size of a maximum induced matching of $G[V_w, \overline{V_w}]$. The `mim`-width of (T, δ) is the maximum `mim`-value over all cuts $(V_w, \overline{V_w})$ defined by a node w of T. The `mim`-width of graph G, denoted $mimw(G)$, is the minimum `mim`-width over all branch decompositions (T, δ) of G. As before a *linear branch decomposition* is a branch decomposition where inner nodes of the underlying tree induces a path.

Since a decomposition of $I(F)$ of can be seen also as a decomposition of F, we immediately get from Lemma 1 the following corollary.

Corollary 6. *For any CNF formula F over m clauses, with no clause containing more than t literals, the* `ps`*-width of F is at most* $\min\{m^k + 1, 2^{tk}\}$ *for* $k = \mathtt{mimw}(I(F))$.

Many classes of graphs have intersection models, meaning that they can be represented as intersection graphs of certain objects, i.e. each vertex is associated with an object and two vertices are adjacent iff their objects intersect. The objects used to define intersection graphs usually consist of geometrical objects such as lines, circles or polygons. Many well known classes of intersection graphs have constant `mim`-width, as in the following which lists only a subset of the classes proven to have such bounds in [4,37].

Theorem 7 ([4,37]). *Let G be a graph. If G is a:*
interval graph then $\mathtt{mimw}(G) \leq 1$.
circular arc graph then $\mathtt{mimw}(G) \leq 2$.
k-trapezoid graph then $\mathtt{mimw}(G) \leq k$.
Moreover there exist linear decompositions satisfying the bound, that can be found in polynomial time (for k-trapezoid assume the intersection model is given).

Let us briefly mention the definition of these graph classes. A graph is an interval graph if it has an intersection model consisting of intervals of the real line. A graph is a circular arc graph if it has an intersection model consisting of arcs of a circle. To build a k-trapezoid we start with k parallel line segments $(s_1, e_1), (s_2, e_2), ..., (s_k, e_k)$ and add two non-intersecting paths s and e by joining s_i to s_{i+1} and e_i to e_{i+1} respectively by straight lines for each $i \in \{1, ..., k-1\}$. The polygon defined by s and e and the two line segments $(s_1, e_1), (s_k, e_k)$ forms a k-trapezoid. A graph is a k-trapezoid graph if it has an intersection model consisting of k-trapezoids. See [7] for information about graph classes and their containment relations.

Combining Corollary 6 and Theorem 7 we get the following

Corollary 8. *Let F be a CNF formula containing m clauses with maximum clause-size t. If $I(F)$ is a:*

interval graph then $\texttt{psw}(F) \leq \min\{m+1, 2^t\}$.
circular arc graph then $\texttt{psw}(F) \leq \min\{m^2+1, 4^t\}$.
k-trapezoid graph then $\texttt{psw}(F) \leq \min\{m^k+1, 2^{tk}\}$.

Moreover there exist linear decompositions satisfying the bound, that can be found in polynomial time (for k-trapezoid assume the intersection model is given).

The incidence graphs of formulas are bipartite graphs, which is not the case for the majority of graphs in the above-mentioned graph classes. In the following we show how to extend the results of Corollary 8 to large classes of bipartite graphs. For a graph G and subset of vertices $A \subseteq V(G)$ the bipartite graph $G[A, \overline{A}]$ is the subgraph of G containing all edges of G with exactly one endpoint in A. For any graph G and $A \subseteq V(G)$ we call $G[A, \overline{A}]$ a *bigraph bipartization* of G, and note that G has a bigraph bipartization for each subset of vertices. For a graph class X we define the class of X *bigraphs* as the bipartite graphs H for which there exists $G \in X$ such that H is isomorphic to a bigraph bipartization of G. For example, a bipartite graph H is an *interval bigraph* if there is some interval graph G and some $A \subseteq V(G)$ with H isomorphic to $G[A, \overline{A}]$.

The following result will allow us to lift the results of Corollary 8 from the given graphs to the bigraph bipartizations of the same graphs.

Theorem 9. *Assume that we are given a CNF formula F of m clauses and maximum clause-size t, a graph G, a subset $A \subseteq V(G)$, and (T, δ_G) a (linear) branch decomposition of G of $\texttt{mim}$-width k. If $I(F)$ is connected and isomorphic to $G[A, \overline{A}]$ (thus $I(F)$ a bigraph bipartization of G) then we can in linear time produce a (linear) branch decomposition (T, δ_F) of F having $\texttt{ps}$-width at most* $\min\{m^k+1, 2^{tk}\}$

Proof. Since each variable and clause in F has a corresponding node in $I(F)$, and each node in $I(F)$ has a corresponding node in G, by defining δ_F to be the function mapping each leaf l of T to the variable or clause in F corresponding to the node $\delta_G(l)$, we get that (T, δ_F) is a branch decomposition of F. Consider a cut $(B, \overline{B})$ induced by a node of (T, δ_F). Note that the $\texttt{mim}$-value of $G[B, \overline{B}]$ is at most k. $I(F)$ is connected which means that we have either A or $\overline{A}$ corresponding to the set of variables of F. Assume wlog the former. Thus $C = \overline{A} \cap B \subseteq \texttt{cla}(F)$ are the clauses in B, with $\overline{C} = \texttt{cla}(F) \setminus C$ and $X = A \cap B \subseteq \texttt{var}(F)$ are the variables in B, with $\overline{X} = \texttt{var}(F) \setminus X$. The $\texttt{mim}$-values of $G[C, \overline{X}]$ and $G[\overline{C}, X]$ are at most k, since these are induced subgraphs of $G[B, \overline{B}]$, and taking induced subgraphs cannot increase the size of the maximum induced matching. Hence by Lemma 1, we have $|\texttt{PS}(F_{C,\overline{X}})| \leq |\texttt{cla}(F)|^k + 1$, and likewise we have $|\texttt{PS}(F_{\overline{C},X})| \leq |\texttt{cla}(F)|^k + 1$, with the maximum of these two being the $\texttt{ps}$-value of this cut. Since the $\texttt{ps}$-width of the decomposition is the maximum $\texttt{ps}$-value of each cut the theorem follows. □

Combining Theorems 9 and 7 we immediately get the following.

Corollary 10. *Let F be a CNF formula containing m clauses with maximum clause-size t. If $I(F)$ is a:*
interval bigraph then $\texttt{psw}(F) \leq \min\{m + 1, 2^t\}$.
circular arc bigraph then $\texttt{psw}(F) \leq \min\{m^2 + 1, 4^t\}$.
k-trapezoid bigraph then $\texttt{psw}(F) \leq \min\{m^k + 1, 2^{tk}\}$.
Moreover there exist linear decompositions satisfying the bound.

In the next section we address the question of finding such linear decompositions in polynomial time. We succeed in the case of interval bigraphs, but for circular arc bigraphs and k-trapezoid bigraphs we must leave this as an open problem.

5 Interval bigraphs and formulas having interval orders

We will in this section show that for formulas whose incidence graph is an interval bigraph we can in polynomial time find linear branch decompositions having small $\texttt{ps}$-width. Let us recall the definition of interval ordering. A CNF formula F has an interval ordering if there exists a linear ordering of variables and clauses such that for any variable x occurring in clause C, if x appears before C then any variable between them also occurs in C, and if C appears before x then x occurs also in any clause between them. See Figure 4 for an example.

By a result of Hell and Huang [18] it follows that a formula F has an interval ordering if and only if $I(F)$ is a interval bigraph.

Theorem 11. *Given a CNF formula F over n variables and m clauses each of at most t literals. In time $\mathcal{O}((m + n)mn)$ we can decide if F has an interval ordering (yes iff $I(F)$ is an interval bigraph), and if yes we solve* #SAT *and weighted* MaxSAT *with an additional runtime of $\mathcal{O}(\min\{m^2, 4^t\}(m + n)m)$.*

Proof. Using the characterization of [18] and the algorithm of [26] we can in time $\mathcal{O}((m + n)mn)$ decide if F has an interval ordering and if yes, then we find it. From this interval ordering we build an interval graph G such that $I(F)$ is a bigraph bipartization of G, and construct a linear branch decomposition of G having $\texttt{mim}$-width 1 [4]. From such a linear branch decomposition we get from Theorem 9 that we can construct another linear branch decomposition of F having $\texttt{ps}$-width $\mathcal{O}(m)$. We then run the algorithm of Theorem 5. □

Order:

$x_1\ c_1\ x_2\ x_3\ c_2\ c_3\ x_4\ x_5$

Clauses:

$c_1 = \{x_1, x_2\}$
$c_2 = \{x_2, \overline{x_3}, x_5\}$
$c_3 = \{x_3, \overline{x_4}, x_5\}$

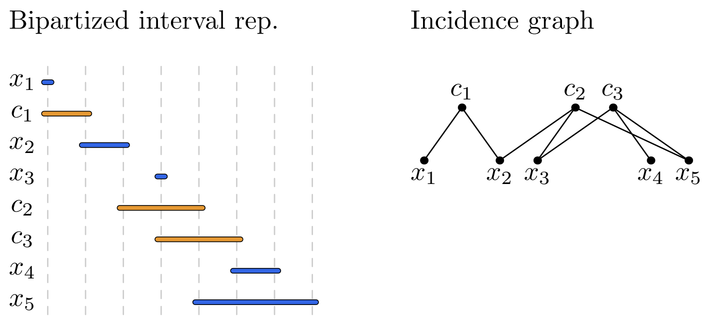


Fig. 4. A CNF formula having an interval ordering. Its incidence graph is an interval bigraph, since it is isomorphic to the bigraph bipartization, defined by the blue intervals, of the interval graph with intersection model on the left.

6 Experimental results

We present some simple experimental results, intended as proof of concept. It is our belief that some of the ideas behind our algorithms, like the notion of ps-value, are useful in practice, but it will require a thorough investigation to confirm such a belief. Our results indicate that the worst-case runtime bounds of the dynamic programming, Theorems 4 and 5, are probably higher than what would commonly be seen in practice.

In the past decade, SAT solvers have become very powerful, and are currently able to handle very large practical instances. Techniques from these SAT solvers have been applied to develop relatively powerful MaxSAT and #SAT solvers [6]. In our experiments we compare implementations of our algorithms against state-of-the-art MaxSAT and #SAT solvers. We do not enhance our implementations with any other techniques, not even simple pre-processing, and on the vast majority of instances our implementations fall far behind in a comparison. However, when focusing on formulas with a certain linear order our implementations compare favorably.

As explained in Section 1, there are two steps involved: (1) find a good decomposition of the input CNF formula F, and (2) perform DP (dynamic programming) along the decomposition. Let us start by describing a very simple heuristic for step (1). It takes as input the bipartite graph $I(F)$ with vertex set $\texttt{cla}(F) \cup \texttt{var}(F)$, and outputs a linear order σ on the vertex set. The below heuristic GreedyOrder is a greedy algorithm that for increasing values of i chooses $\sigma(i)$ to be a vertex having the highest number of already chosen neighbors, and among these choosing one with fewest non-chosen neighbors. This defines a linear branch decomposition (T, δ) of the CNF formula F, with non-leaf nodes of the binary tree T inducing a path, with T rooted at one end of this path, and with δ mapping the ith leaf encountered by a breadth-first search starting at the root of T to the clause or variable $\sigma(i)$, for all $1 \leq i \leq |\texttt{cla}(F) \cup \texttt{var}(F)|$.

```
**Algorithm** GreedyOrder
 **input**: $G = (V, E)$, a (bipartite) graph
**output**: $\sigma$, a linear ordering of $V$

$L = \emptyset, R = V,\ i = 1$
**for all** $v \in V$ set $Ldegree(v) = 0$
**while** $R$ is not empty **do**
    **choose** $v$: from vertices in $R$ with max $Ldegree$ take one of smallest degree
    set $\sigma(i) = v$, increment $i$, add $v$ to $L$ and remove $v$ from $R$
    **for all** $w \in R$ with $vw \in E$ increment $Ldegree(w)$
```

All our implementations can be found online [1]. We have implemented GreedyOrder in Java, together with a straight-forward implementation of the DP algorithm of Theorem 5.

Given a CNF formula, this allows us to solve MaxSAT and #SAT by first running GreedyOrder and then the DP. We compare our implementation to the best solvers we could find online, respectively CCLS-to-akmaxsat [22] which was among the best solvers of the MaxSAT Evaluation competition in 2014 [2], and the latest version of the #SAT solver called sharpSAT developed by Marc Thurley [35,36]. These solvers handily beat our implementation on most inputs. We have therefore generated some CNF formulas having interval orderings, as in Theorem 11, to check if at least on these instances we do better. Note that for step (1) we have not implemented the polynomial-time algorithm recognizing formulas having interval orders, relying instead on the GreedyOrder heuristic.

6.1 Generation of instances

Before presenting our results, let us describe the generation of the set of instances, which are of three types. We start with type 1. The generation of these formulas is based on the definition of interval orderings given by the interval bigraph definition, see e.g. the left side of Figure 4. To generate a formula of type 1 with n variables and m clauses, we generate $n + m$ intervals of the real line by iterating through points i from 1 to $2(n + m)$ as left and right endpoints of the intervals:

- At step i, check which of the 4 cases below are legal (e.g. 3 is legal if there exists a live variable, i.e. with left endpoint $< i$ and no right endpoint) and randomly make one of those legal choices:
  1. start interval of new variable with left endpoint i
  2. start interval of new clause with left endpoint i

3. end interval of randomly chosen live variable by right endpoint i
4. end interval of randomly chosen live clause by right endpoint i

Towards the end of the process boundary conditions are enforced to reach exactly m clauses, with n expected to be slightly smaller than m. For each clause interval we randomly choose each variable having overlapping interval as being either positive or negative in this clause. The resulting CNF formula will have an interval ordering given by the rightmost endpoints of intervals. To hide this ordering the clauses and variables are randomly permuted to make the final CNF formula.

The formulas of type 2 are generated in a very similar fashion as type 1, except we guarantee that all clauses have the same size t, which by Lemma 1 could be of big help. The only change is to case 4 above which instead of being a choice becomes enforced for a live clause that at step i has accumulated exactly t overlapping variable intervals. We also let each clause interval represent 4 clauses over the same variable set but on randomly chosen literals, at the aim of increasing the probability of each instance not being satisfiable.

The formulas of type 3 are the CNF-representation of a conjunction of XOR functions where each XOR has a fixed number t of literals and the variables of the XOR functions overlap in such a way that the incidence graph will be the bipartization of a circular arc graph.

A formula of type 3 is generated from three input parameters n, t, s. It has n variables represented by successive points 1 to n on the circle. The first XOR function has interval from 1 to t thus containing variables with points 1 to t, the second has interval $s+1$ to $s+t$, and in general the ith has interval $i*s+1$ to $i*s+t$, with appropriate modulo addition and some boundary condition at the end to ensure n/s XOR functions. Variables are chosen randomly to appear positive or negative in each XOR. Each XOR is then transformed in the standard way to a CNF formula with 2^{t-1} clauses to give us a resulting CNF formula with $n/s*2^{t-1}$ clauses. Again, variables and clauses are randomly permuted to hide the ordering giving the circular arc bigraph representation.

Note that all the resulting formulas have a quite simple structure, and that a state-of-the-art SAT solver, like `lingeling` [5], handles all generated instances within a few seconds.

6.2 Results

We are now ready to present our results. We ran all the solvers on a Dell Optiplex 780 running Ubuntu 12.04 64-Bit. The machine has 8GB of memory and an Intel Core 2 Quad Q9650 processor with OpenJDK java 6 (IcedTea6 1.13.5).

For instances of type 1 the GreedyOrder heuristic fails terribly and becomes a huge bottleneck. The greedy choice based on degrees of vertices in $I(F)$ is too simple. However, when given the correct interval order to our solver(s) they performed better.

Instances of type 2 are generated similar to those of type 1 but all clauses have small size, which by Lemma 1 could be of help. In this case the number of clauses is approximately four times the number of variables, and as a consequence a great number of the instances were not satisfiable, making the work of the #SAT-solvers easier than that of the MaxSAT solvers. All generated instances of type 2 were solved within seconds by `sharpSAT`, see Figure 5. As the size of the instances grow, we see a clear tendency for the runtimes of `CCLS_to_akmaxsat` to increase much more rapidly than both our solvers. The runtimes of our two solvers were almost identical. The GreedyOrder heuristic on these instances seems to produce decompositions/orders of low PS-width.

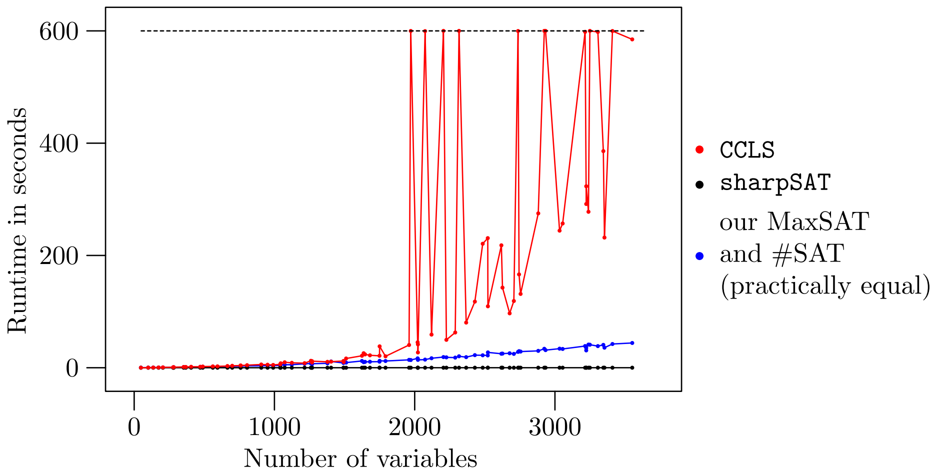


Fig. 5. Runtimes of instances of type 2. Here our MaxSAT solver is clearly faster than `CCLS_to_akmaxsat`. The vertical axis represents time in seconds. Runs taking more than 600 seconds were stopped before completion and are drawn on the dotted line.

The type 3 instances shown in Figure 6 were generated with $k = 5$ and $s = 3$. All instances are satisfiable, which may explain why `CCLS_to_akmaxsat` is very fast. Choosing $k = 3$ and $s = 2$ there will be some not satisfiable instances and `CCLS_to_akmaxsat` would then often spend more than 600 seconds and time out. As the size of the instances grow, we see a clear tendency for the runtimes of `sharpSAT` to increase much more rapidly than our solvers. The runtimes of our two solvers were almost identical.

7 Conclusion

In this paper we have proposed a structural parameter of CNF formulas, called `ps`-width or projection-satisfiable-width. We showed that weighted MaxSAT and #SAT can be solved in polynomial time if given a decomposition of the formula of polynomially bounded `ps`-width. Using the concept of interval bigraphs we also showed a polynomial time algorithm that actually finds such a decomposition, for

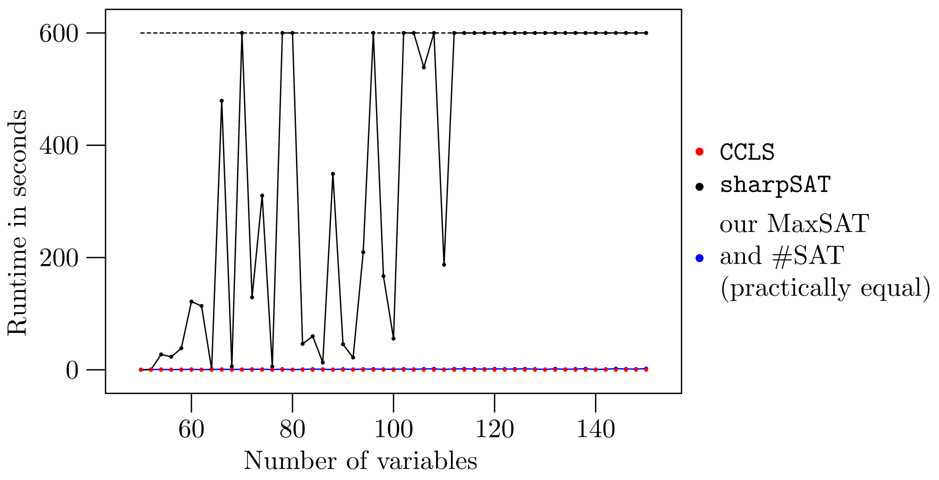


Fig. 6. Runtimes of instances of type 3. Here our #SAT solver is clearly faster than `sharpSAT`. The vertical axis represents time in seconds. Runs taking more than 600 seconds were stopped before completion and are drawn on the dotted line.

formulas having an interval ordering. Could one devise such an algorithm also for the larger class of circular arc bigraphs, or maybe even for the even larger class of k-trapezoid bigraphs? In other words, is the problem of recognizing if a bipartite input graph is a circular arc bigraph, or a k-trapezoid bigraph, polynomial-time solvable?

It could be of practical interest to design a heuristic algorithm which given a formula finds a decomposition of relatively low `ps`-width, as has been done for boolean-width in [19]. One could then check if benchmarks covering real-world SAT instances have low `ps`-width, and perform a study on the correlation between low `ps`-width and their practical hardness by MaxSAT and #SAT solvers, as has been done for treewidth and SAT solvers [23]. We presented some simple experimental results, but it will require a thorough investigation to check if ideas from our algorithms could be useful in practice. Finally, we hope the essential combinatorial result enabling the improvements in this paper, Lemma 1, may have other uses as well.

References

1. `http://people.uib.no/ssa032/pswidth/`
2. Ninth Max-SAT Evaluation (Max-SAT 2014) (2014), `http://www.maxsat.udl.cat/14/`, accessed 16-January-2015
3. Bacchus, F., Dalmao, S., Pitassi, T.: Algorithms and complexity results for#SAT and bayesian inference. In: Foundations of Computer Science, 2003. Proceedings. 44th Annual IEEE Symposium on. pp. 340–351. IEEE (2003)
4. Belmonte, R., Vatshelle, M.: Graph classes with structured neighborhoods and algorithmic applications. Theor. Comput. Sci. 511, 54–65 (2013)

5. Biere, A.: Yet another local search solver and lingeling and friends entering the SAT competition 2014. SAT Competition 2014 p. 39 (2014)
6. Biere, A., Heule, M., van Maaren, H.: Handbook of satisfiability, vol. 185, chap. 20. IOS Press (2009)
7. Brandstädt, A., Le, V.B., Spinrad, J.P.: Graph Classes: A Survey, Monographs on Discrete Mathematics and Applications, vol. 3. SIAM Society for Industrial and Applied Mathematics, Philadelphia (1999)
8. Brandstädt, A., Lozin, V.V.: On the linear structure and clique-width of bipartite permutation graphs. Ars Comb. 67 (2003)
9. Brault-Baron, J., Capelli, F., Mengel, S.: Understanding model counting for β-acyclic CNF-formulas. CoRR abs/1405.6043 (2014), `http://arxiv.org/abs/1405.6043`
10. Bui-Xuan, B.M., Telle, J.A., Vatshelle, M.: H-join decomposable graphs and algorithms with runtime single exponential in rankwidth. Discrete Applied Mathematics 158(7), 809–819 (2010)
11. Bui-Xuan, B.M., Telle, J.A., Vatshelle, M.: Boolean-width of graphs. Theoretical Computer Science 412(39), 5187–5204 (2011)
12. Courcelle, B.: Clique-width of countable graphs: a compactness property. Discrete Mathematics 276(1-3), 127–148 (2004), `http://dx.doi.org/10.1016/S0012-365X(03)00303-0`
13. Darwiche, A.: Recursive conditioning. Artificial Intelligence 126(1), 5–41 (2001)
14. Fischer, E., Makowsky, J.A., Ravve, E.V.: Counting truth assignments of formulas of bounded tree-width or clique-width. Discrete Applied Mathematics 156(4), 511–529 (2008)
15. Fredkin, E.: Trie memory. Communications of the ACM 3(9), 490–499 (1960)
16. Ganian, R., Hlinený, P., Obdrzálek, J.: Better algorithms for satisfiability problems for formulas of bounded rank-width. Fundam. Inform. 123(1), 59–76 (2013)
17. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman (1979)
18. Hell, P., Huang, J.: Interval bigraphs and circular arc graphs. Journal of Graph Theory 46(4), 313–327 (2004)
19. Hvidevold, E.M., Sharmin, S., Telle, J.A., Vatshelle, M.: Finding good decompositions for dynamic programming on dense graphs. In: Marx, D., Rossmanith, P. (eds.) IPEC. Lecture Notes in Computer Science, vol. 7112, pp. 219–231. Springer (2011)
20. Jaumard, B., Simeone, B.: On the complexity of the maximum satisfiability problem for Horn formulas. Inf. Process. Lett. 26(1), 1–4 (1987)
21. Kaski, P., Koivisto, M., Nederlof, J.: Homomorphic hashing for sparse coefficient extraction. In: Proceedings of the 7th international conference on Parameterized and Exact Computation. pp. 147–158. Springer-Verlag (2012)
22. Luo, C., Cai, S., Wu, W., Jie, Z., Su, K.: CCLS: An efficient local search algorithm for weighted maximum satisfiability. IEEE Transactions on Computers (2014)
23. Mateescu, R.: Treewidth in industrial SAT benchmarks. Tech. rep., Tech. rep. Cambridge, UK: Microsoft Research (2011), `http://research.microsoft.com/pubs/145390/MSR-TR-2011-22.pdf`
24. Müller, H.: Recognizing interval digraphs and interval bigraphs in polynomial time. Discrete Applied Mathematics 78(1-3), 189–205 (1997)
25. Paulusma, D., Slivovsky, F., Szeider, S.: Model counting for CNF formulas of bounded modular treewidth. In: Portier, N., Wilke, T. (eds.) STACS. LIPIcs, vol. 20, pp. 55–66. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2013)
26. Rafiey, A.: Recognizing interval bigraphs by forbidden patterns. CoRR abs/1211.2662 (2012)

27. Raman, V., Ravikumar, B., Rao, S.S.: A simplified NP-complete MAXSAT problem. Inf. Process. Lett. 65(1), 1–6 (1998)
28. Rao, M.: Clique-width of graphs defined by one-vertex extensions. Discrete Mathematics 308(24), 6157–6165 (2008)
29. Robertson, N., Seymour, P.D.: Graph minors X. obstructions to tree-decomposition. J. COMBIN. THEORY SER. B 52(2), 153–190 (1991)
30. Roth, D.: A connectionist framework for reasoning: Reasoning with examples. In: Clancey, W.J., Weld, D.S. (eds.) AAAI/IAAI, Vol. 2. pp. 1256–1261. AAAI Press / The MIT Press (1996)
31. Sæther, S.H., Telle, J.A., Vatshelle, M.: Solving MaxSAT and #SAT on structured CNF formulas. In: Sinz, C., Egly, U. (eds.) SAT 2014. Lecture Notes in Computer Science, vol. 8561, pp. 16–31. Springer (2014), `http://dx.doi.org/10.1007/978-3-319-09284-3_3`
32. Samer, M., Szeider, S.: Algorithms for propositional model counting. J. Discrete Algorithms 8(1), 50–64 (2010)
33. Slivovsky, F., Szeider, S.: Model counting for formulas of bounded clique-width. In: Cai, L., Cheng, S.W., Lam, T.W. (eds.) ISAAC. Lecture Notes in Computer Science, vol. 8283, pp. 677–687. Springer (2013)
34. Szeider, S.: On fixed-parameter tractable parameterizations of SAT. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. Lecture Notes in Computer Science, vol. 2919, pp. 188–202. Springer (2003)
35. Thurley, M.: sharpSAT, `https://sites.google.com/site/marcthurley/sharpsat`, accessed 16-January-2015
36. Thurley, M.: sharpSAT–counting models with advanced component caching and implicit BCP. In: Theory and Applications of Satisfiability Testing-SAT 2006, pp. 424–429. Springer (2006)
37. Vatshelle, M.: New width parameters of graphs. Ph.D. thesis, The University of Bergen (2012)