Electronic Colloquium on Computational Complexity, Report No. 3 (2003)

DPLL with Caching
A new algorithm for #SAT and Bayesian Inference

Fahiem Bacchus Shannon Dalmao Toniann Pitassi

Department of Computer Science
University of Toronto
Toronto, Ontario
Canada, ON M5S 3G4
[fbacchus|sdalmao|toni]@cs.toronto.edu

December 19, 2002

Abstract

Bayesian inference and counting satisfying assignments are important problems with numerous applications in probabilistic reasoning. In this paper, we show that plain old DPLL equipped with memoization can solve both of these problems with time complexity that is at least as good as all known algorithms. Furthermore, DPLL with memoization achieves the best known time-space tradeoff. Although their worst case time complexity is no better, our DPLL based algorithms have the potential to achieve much better performance than known algorithms on problems which possess additional structure. Probabilistic models of real situations tend to have such additional structure.

ISSN 1433-8092

1 Introduction

Bayesian inference (BAYES) is an important and well-studied problem with numerous practical applications in probabilistic reasoning. A great deal of research has gone toward understanding this problem, and developing both approximate and exact algorithms for it (e.g., [Pea88, Hec93, Dec96, BFGK96, PP91, Dar01]).

In this paper, inspired by the importance of BAYES, we address the closely related problem #SAT. The decision versions of both #SAT and BAYES are #P-complete [Val79a, Val79b, Rot96]. Furthermore, there are *natural* polynomial-time reductions from each problem to the other [LPI01]. These reductions preserve much of the problem structure, and thus algorithmic insights for one problem can readily be translated into algorithmic insights for the other. In fact, there exists a more general problem (described in Section 4) of which both BAYES and #SAT are instances. It is not difficult to recast the standard algorithms for BAYES and #SAT (including those presented here) into algorithms for this general problem, and thus automatically obtain a way of using these algorithms for either problem. This in itself is an interesting exercise, as algorithms for BAYES have mainly focused on exploiting the structure of problem instances (so that their run times are a function of structural properties of the input), whereas algorithms for #SAT have tended to focus on exploiting the structure of problem classes (so that they yield worst case guarantees on, e.g., all problems in the class 2-CNF). Thus algorithms for BAYES yield new algorithms for #SAT with different performance guarantees.

The main contribution of the paper, however, comes from a different perspective: viewing #SAT as a generalization of the simpler problem SAT. Here our motivation comes from the phenomenal performance exhibited by recent DPLL [DLL62] SAT solvers on structured problems arising from real domains. For example, when run on SAT encodings of bounded model checking problems [BCC$^+$99], the most recent DPLL based SAT solvers, e.g, Zchaff [MMZ$^+$01], can regularly solve *unsatisfiable* instances involving 50,000 variables and 200,000 clauses in less than a 1,000 seconds. So the natural question is whether or not some of this performance can be preserved when we generalize DPLL to solve #SAT. If so, then the potential to significantly improve our ability to solve BAYES exists, either by translating BAYES to #SAT, or by lifting DPLL to solve the more general problem and thus solving BAYES directly.

We first show that the obvious extension of DPLL to solve #SAT can perform very badly in comparision with standard BAYES algorithms (recast to solve #SAT). Examining why the obvious extension of DPLL fails, reveals that much of its computation is redundant: the same computation can be repeated exponentially many times. Caching previous computations is an obvious solution, but there are a number of choices available as to what to cache. We show that different caching schemes produce quite complex differences in the behavior of DPLL, and yield different performance guarantees. Nevertheless, we are able to show that with caching DPLL can perform surprisingly well. In fact, one of our main results is to describe a relatively simple caching scheme that allows DPLL to polynomially-simulate all of the known state-of-the-art BAYES algorithms with respect to time.

Furthermore, our DPLL based algorithm retains the potential of standard DPLL to perform much better than its worst case guarantees on problems that arise from real domains. In particular, it can automatically take advantage of structure in the problem instance that would mostly be ignored by the standard BAYES algorithms.

An outline of the paper follows. In Section 2, we define #SAT, discuss standard DPLL and DPLL adapted to #SAT, and present the graph theoretic notion of branch width that we will use to characterize problem structure. In Section 3, we describe algorithms for #SAT based on different caching schemes for DPLL, and we give a complexity analysis of our algorithms in terms of the branch width of the underlying hypergraph. In Section 4 we show how our algorithms apply not only to #SAT but also to BAYES. To do this, we show that both BAYES and #SAT are instances of a more general problem, and we describe how to recast our DPLL algorithms (as well as known algorithms for BAYES) to solve this more general problem. In

Table 1 Standard DPLL algorithm for SAT

DPLL(ϕ)
    if ϕ has no clauses, output "*satisfiable*" and HALT
    else-if ϕ does not contain an empty clause then
        choose a variable x that appears in ϕ
        Call DPLL($\phi|_{x=0}$)
        Call DPLL($\phi|_{x=1}$)
    return

Section 5, we compare our algorithms to well-known algorithms for BAYES. In Section 6, we discuss some advantages of our algorithms over previous ones, and finally we conclude in Section 7 with open problems and related work.

2 Background

2.1 SAT and #SAT

The SAT decision problem is defined as follows. Let $\mathbf{x} = \langle x_1, x_2, \ldots, x_n \rangle$ be a collection of n Boolean (0/1) variables, and let $\phi(\mathbf{x})$ be a k-CNF Boolean formula on these variables with m clauses.[1] An assignment α to the Boolean variables $\mathbf{x}$ is *satisfying* if it makes the formula `True` ($\phi(\alpha) = 1$) and *unsatisfying* if it makes the formula `False` ($\phi(\alpha) = 0$). The decision problem SAT asks, given a Boolean formula $\phi(\mathbf{x})$ in k-CNF, does it have a satisfying assignment?

The #SAT search problem asks, given a Boolean formula $\phi(\mathbf{x})$ in k-CNF, how many of its assignments are satisfying? Thus #SAT is concerned not just with the existence of a satisfying assignment, but with the *number* of satisfying assignments. Valiant [Val79a, Val79b] showed that the decision version of #SAT is #P hard even when the clause size, k, is 2. Roth [Rot96] showed that the problem is hard even to approximate in many important cases. For example, it is hard to approximate even when $\phi(\mathbf{x})$ is monotone, or Horn, or 2-CNF.

2.2 DPLL and #DPLL

DPLL is the most popular complete algorithm for SAT. Variations of DPLL have recently been applied to solve generalizations of SAT (including #SAT) [Dub91, Zha96, BL99, BP00, LMP00]. The standard DPLL algorithm for solving SAT is given in Table 1. We use the notation $\phi|_{x=0/1}$ to denote the new CNF formula obtained from reducing ϕ by setting the variable x to 0 or 1.

A *decision tree* over binary variables $x_1, \ldots, x_n$ is a binary tree where each internal node in the tree is labeled with a variable x_i such that each variable appears at most once on each path in the tree. The two outgoing edges of each node are labeled with the two different settings of the variable x_i labeling the node: $x_i = 0$ and $x_i = 1$, and each leaf of the tree is labeled with either 0 or 1. Note that any total truth assignment α to the underlying variables is consistent with exactly one path of T. A decision tree T over $x_1, \ldots, x_n$ represents a CNF formula $\phi(x_1, \ldots, x_n)$ if for every total truth assignment α, the value of the leaf node on the path consistent with α is equal to $\phi(\alpha)$. Notice that a given ϕ will in general have many decision trees of varying sizes that can represent it.

[1] For simplicity, and without loss of generality, we assume that we are dealing with k-CNF, for some constant k, rather than clauses of arbitrary length.

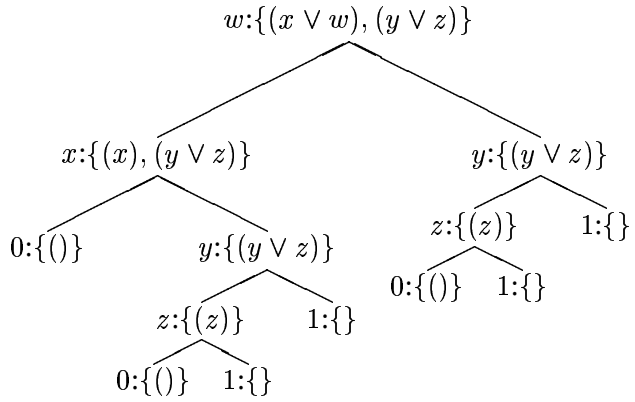

Figure 1: A decision tree for the formula $\{(w \vee x), (y \vee z)\}$

Table 2 DPLL algorithm for computing #SAT

#DPLL(ϕ)
Returns the probability of ϕ
if ϕ has no clauses, return 1
else-if ϕ has an empty clause, return 0
else
Choose a variable x that appears in ϕ
return #DPLL($\phi|_{x=0}$) $\times \frac{1}{2}$ + #DPLL($\phi|_{x=1}$) $\times \frac{1}{2}$

Figure 1 gives an example of a decision tree for $\{(w \vee x), (y \vee z)\}$. Each vertex shows the variable labeling it and, after the ':', the subformula being solved in this subtree. The left branch from a vertex always corresponds to setting the vertex's variable to `False`, while the right branch corresponds to setting the variable to `True`. DPLL for SAT is an algorithm for creating a decision tree (using various heuristics for choosing the next variable to split on) in a depth-first manner, and halting whenever the first leaf labelled by "1" is found.

A slight modification of DPLL allows it to count all satisfying assignments as it traverses the entire decision tree. Table 2 gives the #DPLL algorithm for counting. The algorithm actually computes the probability of the set of satisfying assignments under the uniform distribution. Hence, the number of satisfying assignments can be obtained by simply multiplying by 2^n, where n is the number of variables in ϕ. For a particular formula, solving SAT can take much less time than counting all assignments, since we can terminate the algorithm in the former case whenever the first satisfying assignment is found. However, the same exponential worst-case time bounds apply to both DPLL and to #DPLL. For unsatisfiable formulas, both algorithms have to traverse an entire decision tree before terminating. Thus, we need to exhibit an infinite sequence of unsatisfiable formulas such that any decision tree for these formulas has exponential size. It is implicit in [Hak85, BP96] that *any* decision tree for the formulas encoding the (negation of the) propositional pigeonhole principle have exponential size, and thus DPLL and #DPLL take exponential-time on these examples.

This lower bound does not, however, help us discriminate between algorithms, since all known algorithms for #SAT and BAYES run in exponential-time in the worst case. More useful bounds arise from considering structural properties of the input problem. A method for characterizing structure in the input problem is presented next.

2.3 Complexity measures and tree-width

Associated with every CNF formula ϕ is an underlying hypergraph $\mathcal{H} = (V, E)$, where V is the set of variables appearing in ϕ and each clause generates a hyperedge in E containing the variables of the clause. The "width" of this hypergraph is the critical measure of complexity for essentially all state-of-the-art algorithms for #SAT and BAYES.

There are three different (and well known) notions of width that we will define in this section. We will also show that these different notions of width are basically equivalent. These equivalences are known, although we need to state them and prove some basic properties between them, in order to analyze our new algorithms, and to relate them to standard algorithms.

Definition 1 *Let $\mathcal{H} = (V, E)$ be a hypergraph. A **branch decomposition** for $\mathcal{H}$ is a binary tree T such that each node of T is labelled with a subset of V. There are $|E|$ many leaves of T, and their labels are in one-to-one correspondence with the hyperedges E. For any other node n in T, let A denote the union of the leaf labellings of the subtree rooted at n, and let B denote the union of the labellings of the rest of the leaves. Then the label for n is the set of all vertices v that are in the intersection of A and B. The **branch width** of a branch decomposition T for $\mathcal{H}$ is the maximum size of any labelling in T. The **branch width** of $\mathcal{H}$ is the minimum branch width over all branch decompositions of $\mathcal{H}$.*

Example 1 *Figure 2 shows a particular branch decomposition T_{bd} for the hypergraph $\mathcal{H} = (V, E)$ where $V = \{1, 2, 3, 4, 5\}$ and $E = \{(1, 2, 3), (1, 4), (2, 5), (3, 5), (4, 5)\}$. T_{bd} has branch width 3.*

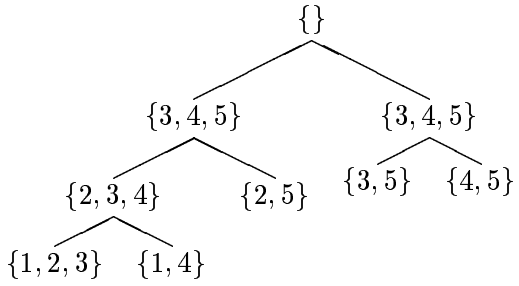

Figure 2: A branch decomposition of width 3 for $\mathcal{H} = \{(1, 2, 3), (1, 4), (2, 5), (3, 5), (4, 5)\}$

Definition 2 *Let $\mathcal{H} = (V, E)$ be a hypergraph, and let $\pi = v_1^\pi, \ldots, v_n^\pi$ be an ordering of all of the vertices in V, where v_i^π is the i^{th} element in the ordering. This induces a sequence of hypergraphs $\mathcal{H}_1, \mathcal{H}_2, \ldots, \mathcal{H}_n$ where $\mathcal{H} = \mathcal{H}_1$ and $\mathcal{H}_{i+1}$ is obtained from $\mathcal{H}_i$ as follows. All edges in $\mathcal{H}_i$ containing v_i^π are merged into one edge and then v_i^π is removed. Thus the underlying vertices of $\mathcal{H}_{i+1}$ are $v_{i+1}^\pi, \ldots v_n^\pi$. The **induced width** of $\mathcal{H}$ under π is the size of the largest edge in all the hypergraphs $\mathcal{H}_1, \ldots, \mathcal{H}_n$. The **elimination width** of $\mathcal{H}$ is the minimum induced width over all orderings π.*

Example 2 *Under the ordering $\pi = \langle 1, 2, 3, 4, 5\rangle$ the hypergraph $\mathcal{H}$ of Example 1 produces the following sequence of hypergraphs:*

$$\begin{aligned}
\mathcal{H}_1 &= \{(1, 2, 3), (1, 4), (2, 5), (3, 5), (4, 5)\} \\
\mathcal{H}_2 &= \{(2, 3, 4), (2, 5), (3, 5), (4, 5)\} \\
\mathcal{H}_3 &= \{(3, 4, 5), (3, 5), (4, 5)\} \\
\mathcal{H}_4 &= \{(4, 5), (4, 5)\} \\
\mathcal{H}_5 &= \{(5)\}
\end{aligned}$$

The induced width of $\mathcal{H}$ under π is 3—the edges $(1,2,3) \in \mathcal{H}_1$, $(2,3,4) \in \mathcal{H}_2$ and $(3,4,5) \in \mathcal{H}_3$ all achieve this size.

Tree width is the third notion of width. Unlike branch and elimination widths, tree width is defined over ordinary graphs, not hypergraphs. But it is easy to reduce a hypergraph to a graph by replacing each hyperedge with a clique of ordinary edges.

Definition 3 *Let $\mathcal{H} = (V, E)$ be a hypergraph. Then the **moralized graph** or primal graph, $G_{\mathcal{H}} = (V', E')$ corresponding to $\mathcal{H}$ is as follows. First, $V' = V$, and secondly, an edge (i, j) is in E' if and only if i and j occur together in some edge E of $\mathcal{H}$.*

Definition 4 *Let $G = (V, E)$ be an undirected graph. A **tree decomposition** of G is a binary tree T such that each node of T is labelled with a subset of V in the following way. First, for every edge $(i, j) \in E$, some leaf node in T must have a label that contains both i and j. Secondly, given labels for the leaf nodes every internal node n contains $v \in V$ in its label if and only if n is on a path between two leaf nodes l_1 and l_2 whose labels contain v. The **tree width** of a tree decomposition T for G is the maximum size of any labelling in T, and the **tree width** of G is the minimum tree width over all tree decompositions of G.*

Example 3 *The moralized graph for $\mathcal{H}$ of Example 1 is the graph $G_{\mathcal{H}} = (V, E)$ with $V = \{1,2,3,4,5\}$ and $E = \{(1,2),(1,3),(1,4),(2,3),(2,5),(3,5),(4,5)\}$. Note that every non-binary hyperedge h of $\mathcal{H}$ yields a clique of edges over the verticies of h in $G_{\mathcal{H}}$. Figure 3 shows T_{td} a tree decomposition for $G_{\mathcal{H}}$. T_{td} has tree width 4.*

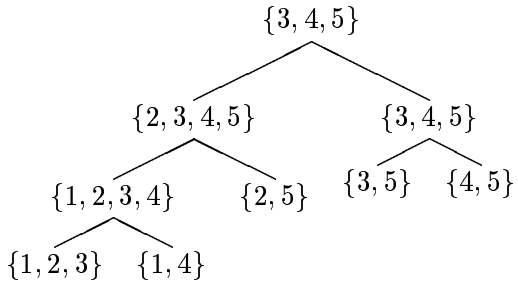

Figure 3: Tree decomposition for $G_{\mathcal{H}}$

The next three lemmas show that these three notions are basically equivalent.

Lemma 1 (Robertson and Seymour [RS91]) *Let $\mathcal{H}$ be a hypergraph and let $G_{\mathcal{H}}$ be the corresponding moralized graph. Then the branch width of $\mathcal{H}$ is at most the tree width of $G_{\mathcal{H}}$ plus 1, and the tree width of $G_{\mathcal{H}}$ is at most 2 times the branch width of $\mathcal{H}$.*

Lemma 2 *Let $\mathcal{H} = (V, E)$ be a hypergraph with a branch decomposition of width w. Then there is an ordering π of the vertices V such that the induced width of $\mathcal{H}$ under π is at most $2w$.*

Proof: Let $\mathcal{H} = (V, E)$ be a hypergraph of branch width w and let T_{td} be a tree decomposition of the moralized graph of $\mathcal{H}$, with width at most $2w$. We can assume without loss of generality that the labels of the leaves of T_{td} are in a one-to-one correspondence with the edges of $\mathcal{H}$. For an arbitrary node m in T_{td}, let $label(m)$ be the set of vertices in the label of m, A^m be the tree rooted at m, $vertices(m)$ be the union of the labels of the leaf nodes in A^m (i.e., the hyperedges of $\mathcal{H}$ appearing below A^m), and $depth(m)$ be the distance from m to the root.

Let x be any vertex of $\mathcal{H}$, and let $leaves(x)$ be the set of leaves of T_{td} that contain x in their label. We define $node(x)$ to be the deepest common ancestor in T_{td} of all the nodes in $leaves(x)$, and the depth of a vertex, $depth(x)$, to be $depth(node(x))$. Note that $x \in label(node(x))$, since the path from the left-most leaf in $leaves(x)$ to the right-most leaf must pass through $node(x)$; and that x does not appear in the label of any node outside of the subtree rooted at $node(x)$, since no leaf outside of this subtree contains x.

Finally let $\pi = x_1, \ldots, x_n$ be any ordering of the vertices such that if $depth(y) < depth(x)$, then y must follow x in the ordering. We use the notation $y >_\pi x$ to indicate that y follows x in the ordering π (and thus y will be eliminated after x). We claim that the induced width of π is at most the width of T_{td}, i.e., $2w$.

Consider $A^{node(x)}$, the subtree rooted at $node(x)$, and $vertices(node(x))$, the union of the labels of the leaves of $A^{node(x)}$. We make the following observations about these vertices.

1. If $y \in vertices(node(x))$ and $y >_\pi x$, then $y \in label(node(x))$ and $node(y)$ must be an ancestor of $node(x)$ (or equal). $y >_\pi x$ implies that $depth(y) \leq depth(x)$. There must be a path from the leaf in $A^{node(x)}$ containing y to $node(y)$, and since $node(y)$ is at least as high as $node(x)$ the path must go through $node(x)$ (or we must have $node(x) = node(y)$). In either case $y \in label(node(x))$.

2. If $y \in vertices(node(x))$ and $y <_\pi x$ then $node(y)$ must lie inside $A^{node(x)}$ so $node(y)$ must be a descendant of $node(x)$ (or equal). $y <_\pi x$ implies that $depth(y) \geq depth(x)$. There must be a path from the leaf in $A^{node(x)}$ containing y to $node(y)$, and since $node(y)$ is at least as deep as $node(x)$ there must either be a further path from $node(y)$ to $node(x)$, or $node(y) = node(x)$.

Note further that condition 2 implies that if $y <_\pi x$ and y appears in $A^{node(x)}$, the subtree below $node(x)$, then all hyperedges in the original hypergraph $\mathcal{H}$ containing y must also be in $A^{node(x)}$.

We claim that the hyperedge produced at stage i in the elimination process when x_i is eliminated is contained in $label(node(x_i))$. Since the size of this set is bounded by $2w$, we thus verify that the induced width of π is bounded by $2w$.

The base case is when x_1 is eliminated. All hyperedges containing x_1 are contained in the subtree below $node(x_1)$, thus the hyperedge created when x_1 is eliminated is contained in $vertices(node(x_1))$. All other vertices in $vertices(node(x_1))$ follow x_1 in the ordering so by the above they must label $node(x_1)$ and $vertices(node(x_1)) \subseteq label(node(x_1))$.

When x_i is eliminated there are two types of hyperedges that are unioned together: (a) the remaining hyperedges containing x_i that were part of the original hypergraph $\mathcal{H}$, and (b) the remaining hyperedges containing x_i that were produced as $x_1, \ldots, x_{i-1}$ were eliminated. For the original hyperedges, all of these are among the leaves below $node(x_i)$, and thus are contained in $vertices(node(x_i))$. Now consider a new hyperedge produced by eliminating one of the previous variables, say the variable y. This hyperedge is contained in $label(node(y))$ by induction, which in turn is contained in $vertices(node(y))$. Moreover, it must be that $node(y)$ is in $A^{node(x_i)}$. Otherwise, $node(y)$ cannot contain x_i in its label (no node outside the subtree below $node(x_i)$ has x_i in its label), and the hyperedge created when y is eliminated also cannot contain x_i. Since $node(y)$ is in $A^{node(x_i)}$, we get that the hyperedge created when y was eliminated is contained in $vertices(node(x_i))$ since this is a superset of $vertices(node(y))$.

In sum the hyperedge created when x_i is eliminated is contained in $vertices(node(x_i))$, since all of the hyperedges containing x_i at this stage are in this set. Furthermore, all vertices $x_1, \ldots, x_{i-1}$ are removed from this hyperedge, thus it contains only variables following x_i in the ordering. Hence, by (1) above this hyperedge is contained in $label(node(x_i))$. ■

Example 4 *The tree decomposition T_{td} given in Figure 3 has the property that it has tree width no more than twice the branch width for the branch decomposition T_{bd} given in Figure 2. T_{td} gives rise to the ordering $\pi = \langle 1, 2, 3, 4, 5 \rangle$. For example, the deepest common ancestor of the vertex 2 is the node labeled*

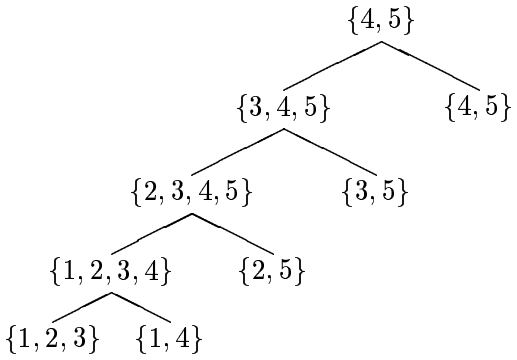

Figure 4: Tree decomposition for $\pi = \langle 1, 2, 3, 4, 5 \rangle$

$(2, 3, 4, 5)$ *(the left child of the root), while the deepest common ancestor of the vertex 1 is the node labled* $(1, 2, 3, 4)$ *(the left child of deepest common ancestor of 2). Hence 2 must follow 1 in the ordering. As shown in Example 2,* π *has induced width 3, which is less than twice the branch width of the branch decomposition* T_{bd}*.*

Lemma 3 *Let* $\mathcal{H}$ *be a hypergraph with elimination width at most* w*. Then the moralized graph* $G_{\mathcal{H}}$ *has a tree-decomposition of tree width at most* $w + 1$*. And hence,* $\mathcal{H}$ *has a branch decomposition of tree width at most* $w + 1$*.*

Proof: Let $\pi = x_1, \ldots, x_n$ be an elimination ordering for $\mathcal{H}$. Then we will construct a tree decomposition for $G_{\mathcal{H}}$, the moralized graph of $\mathcal{H} = (V, E)$, using π as follows. Initially, we have $|E|$ trees, each of size 1, one corresponding to each edge $e \in E$. We first merge the trees containing x_1 into a bigger tree, T_1, leaving us with a new, smaller set of trees. Then we merge the trees containing x_2 into a bigger tree, T_2. We continue in this way until we have formed a single tree, T. Now we fill in the labels for all intermediate vertices of T so that the tree is a tree-decomposition. That is, if m and n are two leaves of T and they both contain some vertex v, then every node along the path from m to n must also contain v in its label. It is not too hard to see that for each x_i, the tree T_i (created when merging the trees containing x_i) has the property that all labels of T_i are contained in $e_i \cup x_i$, where e_i is the hyperedge created when x_i is eliminated. Thus the treewidth of the final tree T can be no larger than the induced width of π plus 1. ■

Example 5 *Figure 4 shows the tree decomposition generated by the elimination ordering* $\pi = \langle 1, 2, 3, 4, 5 \rangle$ *for* $\mathcal{H}$ *from Example 1.* π *has induced width 3 and the resulting tree decomposition has tree width 4.*

2.4 Complexity as a function of branch width

The best algorithms for BAYES (and thus for #SAT) run in time $n2^{O(w)}$ where w is the *branch width* of the underlying hypergraph of the input. We first show that #DPLL, the simple extension of DPLL for #SAT, performs poorly under this metric.

Theorem 4 *There exist families of CNF formulas with underlying branch width 0 such that counting satisfying assignments using #DPLL on these formulas takes exponential time.*

Proof: Consider a 3CNF formula over $3n$ variables consisting of n disjoint clauses. This formula has branch width 0; however, any complete decision tree requires exponential size. Therefore the #DPLL algorithm will require exponential time. ■

Table 3 #DPLL algorithm with simple caching

#DPLLSimpleCache(ϕ)
  If InCache($\{\phi\}$), return
  else
    Pick a variable v in ϕ
    $\phi^- = \phi|_{v=0}$
    #DPLLSimpleCache(ϕ^-)
    $\phi^+ = \phi|_{v=1}$
    #DPLLSimpleCache(ϕ^+)
    AddToCache(ϕ, GetValue(ϕ^-) $\times \frac{1}{2}$ + GetValue(ϕ^+) $\times \frac{1}{2}$)
    return

3 DPLL with caching

3.1 The algorithms

If one considers the example used to prove Theorem 4 it can be seen that #DPLL's poor performance arises from the fact that during the course of its execution the same subproblem can be encountered and recomputed many times. Our previous example also demonstrates this. Figure 1 shows that a run of #DPLL on $f = \{(w \vee x), (y \vee z)\}$ using this decision tree will encounter the subproblem $\{(y \vee z)\}$ twice: once along the path $(w = 0, x = 1)$ and again along the path $(w = 1)$.

One way to try to prevent this duplication is to apply memoization. More specifically, associated with every node in the DPLL tree is a formula f such that the subtree rooted at this node is trying to compute the number of satisfying assignments to f. When performing a depth-first search of the tree, we can keep a cache that contains all formulas f that have already been solved, and upon hitting a new node of the tree we can avoid traversing its subtree if the value of its corresponding formula is already stored in the cache. In the above example, we will cache $\{(y \vee z)\}$, when we solve it along the path $(w = 0, x = 1)$ thereby truncating the subtree below $(w = 1)$.

The above form of caching, which we will call *simple caching* can be easily implemented as shown in Table 3. On return the value of the input formula has been stored in the cache, so a call to GetValue(ϕ) will return the desired value.[2]

In addition to formulas stored in the cache there are also the following *obvious* formulas whose values are easy to compute:

- The empty formula $\{\}$ containing no clauses. Its value is 1.
- Any formula containing the empty clause. All of these formulas have value 0.
- Any formula containing only one clause. If the clause contains j variables its value will be $1 - \frac{1}{2^j}$ (the probability of satisfying the clause).

Obvious formulas need not be stored in the cache, rather their values can be computed as required. We say that a formula is *known* if its value is currently stored in the cache or if it is obvious. If Φ is a set of formulas we assign it a value equal to the product of the values of the formulas in it.[3] We say that Φ is *known* if either (a) all $\phi_i \in \Phi$ are known, or (b) there exists a $\phi_i \in \Phi$ whose value is known to be 0.

The algorithm uses the following (low complexity) subroutines to access its cache.

[2]The cached value is actually the probability of ϕ, so we must multiply it by 2^n to get the number of satisfying assignments.

[3]If ϕ is the conjunction of a set Φ of pairwise disjoint formulas, this value will be the probability of ϕ

Table 4 #DPLL algorithm with caching

#DPLLCache(Φ)
  If InCache(Φ), return
  else
    Φ = RemoveCachedComponents(Φ)
    Pick a variable v in some component $\phi \in \Phi$
    $\Phi^- =$ ToComponents($\phi|_{v=0}$)
    #DPLLCache($\Phi - \{\phi\} \cup \Phi^-$)
    $\Phi^+ =$ ToComponents($\phi|_{v=1}$)
    #DPLLCache($\Phi - \{\phi\} \cup \Phi^+$)
    AddToCache(ϕ, GetValue(Φ^-) $\times \frac{1}{2}$ + GetValue(Φ^+) $\times \frac{1}{2}$)
    return

1. AddToCache(ϕ, r): adds to the cache the fact that formula ϕ has value r.
2. InCache(Φ): takes as input a *set* of formulas Φ and returns true if Φ is known.
3. GetValue(Φ): takes as input a set Φ of known formulas and returns the value of the set (i.e., the product of the values of its formulas).

Surprisingly, simple caching, does reasonably well as the following theorem shows.

Theorem 5 *For solving* #SAT *on n variables, #DPLLSimpleCache runs in time bounded by $2^{O(w \log n)}$ where w is the underlying branch width of the instance.*

The proof of this theorem is given below. Although #DPLLSimpleCache does fairly well, its performance is not quite as good as the best BAYES algorithms (which run in time $n2^{O(w)}$). One of our main contributions is to show that a slightly more sophisticated variant of caching allows #DPLL to perform as well as the best known algorithms.

This version is very similar to #DPLLSimpleCache. The algorithm creates a DPLL tree, as described above, caching intermediate formulas as they are computed. However, the algorithm takes as input formulas that have been decomposed into disjoint components, and the intermediate formulas it caches are similarly stored as disjoint components. Thus, if we have already computed the number of satisfying assignments for f and for g, where f and g are over disjoint sets of variables, we will be able to compute the number of satisfying assignments for $f \wedge g$ without further work.

The new algorithm uses the subroutines previously defined along with two additional (low complexity) subroutines.

4. ToComponents(ϕ): takes as input a formula ϕ, breaks it up into a set of minimal sized disjoint components, and returns this set.
5. RemoveCachedComponents(Φ): returns the input set of formulas Φ with all known formulas removed.

The input to #DPLLCache is a set of disjoint formulas. That is, to run #DPLLCache on the formula ϕ we initially make the call #DPLLCache(ToComponents(ϕ)). When the call #DPLLCache(Φ) returns, the cache will contain sufficient information so that the call GetValue(Φ) will return the desired value.

We will now state our upper bound on the runtime of #DPLLCache. The proof is given below.

Table 5 #DPLL algorithm with caching and linear space

#DPLLSpace(Φ)
  If InCache(Φ), return
  else
    Φ = RemoveCachedComponents(Φ)
    Pick a variable v in some component $\phi \in \Phi$
    $\Phi^- =$ ToComponents($\phi|_{v=0}$)
    #DPLLCache($\Phi - \{\phi\} \cup \Phi^-$)
    $\Phi^+ =$ ToComponents($\phi|_{v=1}$)
    #DPLLCache($\Phi - \{\phi\} \cup \Phi^+$)
    AddToCache(ϕ, GetValue(Φ^-) $\times \frac{1}{2}$ + GetValue(Φ^+) $\times \frac{1}{2}$)
    RemoveFromCache($\Phi^- \cup \Phi^+$)
    return

Theorem 6 *For solving* #SAT *on n variables, #DPLLCache runs in time bounded by $n^{O(1)}2^{O(w)}$ where w is the underlying branch width of the instance.*

Finally, we present a third variant of #DPLL with caching that achieves a nontrivial time-space tradeoff. This algorithm is the natural variant of #DPLLCache, modified to remove cached values so that only linear space is consumed. The only change in the algorithm from #DPLLCache is that it utilizes one additional subroutine:

6. RemoveFromCache(Φ): takes as input a set of formulas (a set of components) and removes all of them from the cache.

After splitting a component with a variable instantiation and computing the value of each part, #DPLLSpace cleans up the cache by removing all of these sub-components, so that only the value of the whole component is retained.

The following theorem shows that #DPLLSpace has time and space complexity that is the same as recursive conditioning [Dar01].

Theorem 7 *For solving* #SAT *on n variables, #DPLLSpace uses only space linear in n and runs in time bounded by $2^{O(w \log n)}$ where w is the underlying branch width of the instance.*

3.2 Proofs of Theorems 5–7

For the proofs of theorems 5 and 6 we will need some common notation and definitions. Let f be k-CNF formula with n variables and m clauses, let $\mathcal{H}$ be the underlying hypergraph associated with f with branch width w. By [Dar01], there is a branch decomposition T_{bd} of $\mathcal{H}$ of depth $O(\log m)$ and width $O(w)$.

Recall that the leaves of T_{bd} are in one-to-one correspondence with the clauses of f. We will number the vertices of T_{bd} according to a depth-first preorder traversal of T_{bd}. For a vertex numbered i, let f_i denote the subformula of f consisting of the conjunction of all clauses corresponding to the leaves of the tree rooted at i. Let $variables(f_i)$ be the set of underlying variables in the (sub)formula f_i. Recall that in a branch decomposition each vertex i of T_{bd} is labelled with $label(i)$, the set of variables in the intersection of $variables(f_i)$ and $variables(f - f_i)$. Each node i in T_{bd} partitions the clauses of f into three sets of clauses: f_i, f_i^L, and f_i^R, where f_i^L is the conjunction of clauses at the leaves of T_{bd} to the left of f_i, and f_i^R is the conjunction of clauses at the leaves to the right of f_i.

All of our DPLL caching algorithms achieve the stated run time bounds by querying the variables in a specific, static order. That is, down any branch of the DPLL decision tree, DT the same variables are instantiated and they are instantiated in the same order. The variable ordering used in DT is determined by the depth-first pre-ordering of the vertices in the branch decomposition T_{bd} and by the labeling of these vertices. Let $(i,1),\cdots,(i,j_i)$ denote the variables in $label(i)$ that do not appear the label of an eariler vertex of T_{bd}. Note that since the width of T_{bd} is w, $j_i \leq w$ for all i. Let $1,\ldots,z$ be the sequence of vertex numbers of T_{bd}. Then our DPLL algorithm will query the variables underlying f in the following static order:

$$\pi = \Big((i_1,1),(i_1,2),\ldots,(i_1,j_1),(i_2,1),\ldots,(i_2,j_2),\ldots,(i_s,1),\ldots,(i_s,j_s)\Big),$$

$i_1 < i_2 < \cdots < i_s \leq z$, and $j_1,\ldots,j_s \leq w$. Note that for some vertices i of T_{bd}, nothing will be queried at this vertex since all of the variables in its label may have occured in the labels of earlier vertices. Our notation allows for these vertices to be skipped. The underlying complete decision tree, DT, created by our DPLL algorithms on input f is thus a tree with $j_1 + j_2 + \cdots + j_s = n$ levels. The levels are grouped into s layers, with the i^{th} layer consisting of j_i levels. Note that there are 2^l nodes at level l in DT, and we will identify a particular node at level l by (l,ρ) where ρ is a particular assignment to the first l variables in the ordering, or by $((q,r),\rho)$, where (q,r) is the l^{th} pair in the ordering π, and ρ is as before.

The DPLL algorithms carry out a depth-first traversal of DT, keeping formulas in the cache that have already been solved along the way. (For #DPLLSimpleCache, the formulas stored in the cache are of the form $f|_\rho$, and for #DPLLCache and #DPLLSpace, the formulas stored are various components of ToComponents$(f|_\rho)$.) If the algorithm ever hits a node where the formula to be computed has already been solved, it can avoid that computation, and thus it does not do a complete depth-first search of DT but rather it does a depth-first search of a *pruned* version of DT. For our theorems, we want to get an upper bound on the size of the pruned tree actually searched by the algorithm.

We will have the following running example to help explain the proofs. Consider what happens when we apply one of our DPLL algorithms to the formula

$$f = \{(x_2 \vee x_3),(x_1 \vee x_3),(x_2 \vee x_4),(x_1 \vee x_4),(x_5 \vee x_6),(x_1 \vee x_6),(x_5 \vee x_7 \vee x_8),(x_7 \vee x_8)\},$$

using the branch decomposition given in Figure 5. The nodes in the figure have two labels. The top label is the number of the node according to the preorder depth-first traversal. The bottom label for node i is $label(i)$, the set of variables in the interesection of $variables(f_i)$ and $variables(f - f_i)$. In this example there are 15 vertices in T_{bd}, but only 8 nonempty layers, one for each of the eight variables. The resulting variable ordering is as follows:

$$\pi = (\underbrace{(2,1)}_{x_1},\underbrace{(3,1)}_{x_2},\underbrace{(4,1)}_{x_3},\underbrace{(7,1)}_{x_4},\underbrace{(10,1)}_{x_5},\underbrace{(11,1)}_{x_6},\underbrace{(14,1)}_{x_7},\underbrace{(14,2)}_{x_8}).$$

This notation explains, for example, that the first variable, x_1, is queried when we reach vertex 2; the second variable, x_2, is queried when we reach vertex 3; and the seventh and eight variables, x_7 and x_8, are not queried until we reach vertex 14 of the branch decomposition.

Theorem 5 *For solving* #SAT *on n variables, #DPLLSimpleCache runs in time bounded by $2^{O(w \log n)}$ where w is the underlying branch width of the instance.*

Proof:

We want to show that the size of the pruned tree (in other words, the size of the subtree of DT actually searched by #DPLLSimpleCache) is at most $2^{O(w \log n)}$.

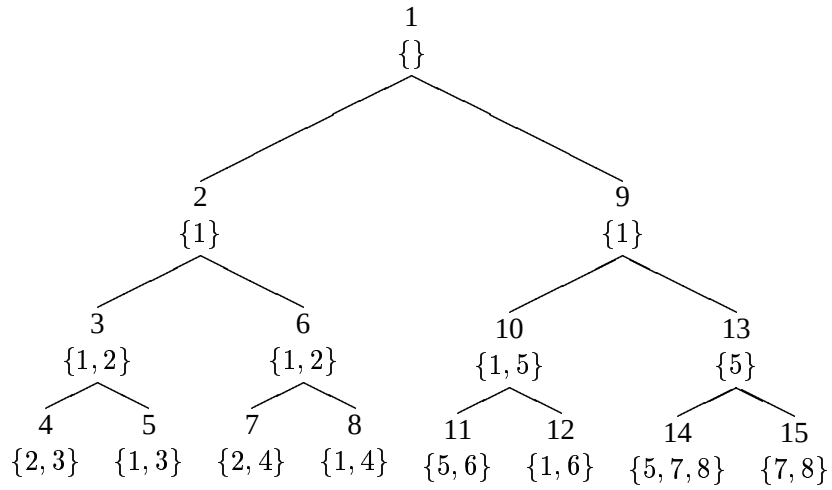

Figure 5: Branch Decomposition for $f = \{(x_2 \vee x_3), (x_1 \vee x_3), (x_2 \vee x_4), (x_1 \vee x_4), (x_5 \vee x_6), (x_1 \vee x_6), (x_5 \vee x_7 \vee x_8), (x_7 \vee x_8)\}$

When backtracking to a particular node $(l, \rho) = ((q, r), \rho)$ at level l in DT, the formula put in the cache, if it is not already known, is of the form $f|_\rho$. (Recall ρ is a setting to the first l variables.) However, we will see that although there are 2^l different ways to set ρ, the number of *distinct* formulas of this form is actually much smaller than 2^l. Consider a partial assignment, ρ, where we have set all variables up to and including (q, r), for some $q \leq i_s$ and some $r \leq j_q$. The number of variables set by ρ (the *length* of ρ) is $j_1 + j_2 + \cdots + j_{q-1} + r$.

Let ρ^- denote the partial assignment that is consistent with ρ where only the variables in ρ that came from the labels of the vertices on the path from the root of T_{bd} up to and including vertex q are set. The idea is that ρ^- is a reduction of ρ, where ρ^- has removed the assignments of ρ that are irrelevant to f_q and f_q^R.

Consider what happens when the DPLL algorithm reaches a particular node $((q, r), \rho)$ at level l of DT. At that point the algorithm is solving the subproblem $f|_\rho$, and thus, once we backtrack to this node, $f|_\rho = f_q^L|_\rho \wedge f_q|_\rho \wedge f_q^R|_\rho$ is placed in the cache, if it is not already known. Note that all variables in the subformula f_q^L are set by ρ, and thus either $f_q^L|_\rho = 0$, in which case nothing new is put in the cache, or $f_q^L|_\rho = 1$ in which case $f|_\rho = f_q|_\rho \wedge f_q^R|_\rho = f_q|_{\rho^-} \wedge f_q^R|_{\rho^-}$ is put in the cache. Thus, the set of *distinct* subformulas placed in the cache at level $l = (q, r)$ is at most the set of all subformulas of the form $f_q|_{\rho^-} \wedge f_q^R|_{\rho^-}$, where ρ^- is a setting to all variables in the labels from the root to vertex q, plus the variables $(q, 1), ..., (q, r)$. There are at most $d \cdot w$ such variables, where q has depth d in T_{bd} (the each label has at most w variables since this is the width of T_{bd}). Hence the total number of such ρ^-'s is at most $2^{(w \cdot d)}$. This implies that the number of subtrees in DT at level $l + 1$ that are actually traversed by #DPLLSimpleCache is at most $2 \cdot 2^{w \cdot d} = 2^{O(w \cdot d)}$, where d is the depth of node q in T_{bd}. Let t be the number of nodes in DT that are actually traversed by #DPLLSimpleCache. Then, t is at most $n2^{O(w \cdot \log n)}$, since t is the sum of the number of nodes visited at every level of DT and for each node q in T_{bd}, $d \in O(\log m) = O(\log n)$.

The overall runtime of #DPLLSimpleCache is at most t^2, where again t is the number of nodes in DT that are traversed by the algorithm. This can be seen by noticing that for every node in DT that is traversed, the algorithm takes time proportional to the size of the cache, which is at most t. Thus, #DPLLSimpleCache runs in time $(n2^{O(w \cdot \log n)})^2 = 2^{O(w \cdot \log n)}$.

In the above running example, consider the set of all 2^5 partial restrictions ρ where we have set all variables up to and including the fifth level. For this case, $f_{10}^L = \{(x_2 \vee x_3), (x_1 \vee x_3), (x_2 \vee x_4), (x_1 \vee x_4)\}$,

$f_{10} = \{(x_5 \vee x_6), (x_1 \vee x_6)\}$ and $f_{10}^R = \{(x_5 \vee x_7 \vee x_8), (x_7 \vee x_8)\}$. f_{10}^L will have vanished by level 5, and While there are 2^5 possible formulas $f|_\rho$, the number of distinct formulas in this set is only 2^2, corresponding to all possible ways to set variables x_1 and x_5.

■

Theorem 6 *For solving* #SAT *on n variables, #DPLLCache runs in time bounded by $n^{O(1)}2^{O(w)}$ where w is the underlying branch width of the instance.*

Proof: We prove the theorem by placing a bound on the number of times #DPLLCache can branch on any variable x_l. Using the notation specified above, x_l corresponds to some pair (q, r) in the ordering π used by #DPLLCache. That is, x_ℓ is the r'th new variable in the label of vertex q of the branch decomposition T_{bd}.

When #DPLLCache utilizes the static ordering π, it branches on, or queries, the variables according to that order, always reducing the component containing the variable x_i that is currently due to be queried. However, since previously cached components are always removed (by RemoveCachedComponents in the algorithm), it can be that when it is variable x_i's turn to be queried, there is no component among the active components (i.e., the components of the current recursive call that remain after all known components are removed) that contains x_i. In this case, #DPLLCache simply moves on to the next variable in the ordering, continuing to advance until it finds the first variable that does appear in some active component. It will then branch on that variable reducing the component it appears in, leaving the other components unaltered.

This implies that at any time when #DPLLCache selects x_l as the variable to next branch on it must be the case that

1. x_l appears in an active component. In particular the value of this component is not already in the cache.

2. No variable prior to x_l in the ordering π appears in any active component. All of these variables have either been assigned a particular value by previous recursive invocations, or the component they appeared in has been removed because its value was already in the cache.

In the branch decomposition T_{bd} let p be q's parent (or q itself if q has no parent). We claim that whenever #DPLLCache selects x_l as the next variable to branch on, the active component containing x_l must be a component that has arisen from a reduction of f_p (the conjunction of the clauses below p), $reduce(f_p)$. In particular, at this stage f_p has been reduced by the assignments made by #DPLLCache's previous recursive invocations, it has been split into components, and all known components have been removed.

First, if $p = q$, then q is the root of T_{bd}, f_p is the original formula f, and the claim is trivial. Otherwise, we know that all of the variables in $label(p)$ have been removed from all active components, since all of these variables must precede x_l in the ordering π. But by the definition of the labels in a branch decomposition, this means that $reduce(f_p)$ can share no variables with any other active component. Hence, at this stage, $reduce(f_p)$ is a set of components disjoint from the other active components. Since x_l is in $label(q)$ and q is in the subtree below p, x_l must be in one of these components if it is in any component.

At this stage, prior to the removal of known components, $reduce(f_p)$ can only be in one of at most $2^{(w+r)}$ different configurations. Of the variables prior to x_l in π only those from $label(p)$ (at most w) and $label(q)$ (at most $r-1$ since x_l is the r'th new variable in $label(q)$) appear in f_p and thus only their settings can affect the form of f_p at this stage. Thus the active component containing x_l can appear in at most $2^{O(w)}$ different configurations and #DPLLCache can branch on x_l at most $2^{O(w)}$ times as each time one more of these configurations gets stored in the cache.

Thus with n variables we obtain a bound on the number of branches in the decision tree explored by #DPLLCache of $n2^{O(w)}$. Clearly, the number of branches explored also bounds the run time.

■

Theorem 7 *For solving* #SAT *on n variables, #DPLLSpace uses only space linear in n and runs in time bounded by $2^{O(w \log n)}$ where w is the underlying branch width of the instance.*

Proof: Let f be a k-CNF formula with n variables and m clauses and let $\mathcal{H}$ be the underlying hypergraph associated with f. Unlike the previous proofs, we will begin with a tree decomposition (rather than a branch decomposition) T_{td} of depth $O(\log m)$ and width $O(w)$. We can assume without loss of generality that the leaves of T_{td} are in one-to-one correspondence with the clauses of f. Each node i in T_{td} partitions f into three disjoint sets of clauses: f_i, the conjunction of clauses at the leaves of the subtree of T_{td} rooted at i, f_i^L, the conjunction of clauses of leaves of T_{td} to the left of f_i, and f_i^R, the conjunction of clauses of leaves of T_{td} to the right of f_i. Let i be a non-leaf node of T_{td} and let j and k denote its left and right children, respectively. The label $label(i)$ associated with node i in T_{td} contains exactly those variables that occur in both f_j and f_k. #DPLLSpace will query the variables associated with the labels of T_{td} according to the depth-first preorder traversal. Let the variables in $label(i)$ not appearing in an earlier label on the path from the root to node i be denoted by $S(i) = (i,1), \ldots, (i, j_i)$. Note that the variables in $S(i)$ are exactly the variables that occur in both f_j and f_k (where j and k are the children of i) but that do not occur outside of f_i. If we let c be the total number of nodes in T_{td}, then #DPLLSpace will query the variables underlying f in the following static order:

$$S(1), S(2), \ldots, S(c).$$

Note that some $S(i)$ may be empty. The underlying decision tree, DT, created by #DPLLSpace is a complete tree with n levels. We will identify a particular node s at level l of DT by $s = (l, \rho)$ where ρ is a particular assignment to the first l variables in the ordering, or by $s = ((q,r), \rho)$. In the latter notation $l = (q,r)$ means that at level l, we query the r^{th} variable in $S(q)$.

We point out that #DPLLSpace as defined earlier, specifies that we must choose the next variable to split on from the set of variables that currently occur in an active component. However, to simplify this proof, we will query the variables in the static ordering as defined above and thus a variable may be queried even if it does not occur in any active component. However it is not hard to see that this can only increase the runtime and does not lead to any real changes in the run of the algorithm. This is because when a variable, x, is queried that is not part of any component of Φ, the sets Φ^- and Φ^+ are unchanged.

#DPLLCache carries out a depth-first traversal of DT, storing the components of formulas in the cache as they are solved. However, now components of formulas are also popped from the cache so that the total space ever utilized is linear. If the algorithm ever hits a node where all of the components of the formula to be computed are known, then the algorithm can avoid traversing the subtree rooted at that node. Thus the algorithm does not do a complete depth-first search of DT but rather it does a depth-first search of a *pruned* version of DT.

During the (pruned) depth-first traversal of DT, each edge that is traversed is traversed twice, once in each direction. At a given time t in the traversal, let $E = E_1 \cup E_2$ be the set of edges that have been traversed, where E_1 are the edges that have only been traversed in the forward direction, and E_2 are the edges that have been traversed in both directions. The edges in E_1 constitute a partial path p starting at the root of DT. Each edge in p is labelled by either 0 or 1. Let $p_1, \ldots, p_k$ be the set of all subpaths of p (beginning at the root) that end in a 1-edge. Let $\rho_1, \ldots, \rho_k$ be subrestrictions corresponding to $p_1, \ldots, p_k$ except that the last variable that was originally assigned a 1 is now assigned a 0. (For example, if p is $(x_1 = 0, x_3 = 1, x_4 =$

$0, x_5 = 1, x_6 = 0, x_2 = 0)$, then $\rho_1 = (x_1 = 0, x_3 = 0)$, and $\rho_2 = (x_1 = 0, x_3 = 1, x_4 = 0, x_5 = 0)$. Then the information that is in the cache at time t contains $ToComponents(f|_{\rho_i})$, $i \leq k$.

For a node q of T_{td} and corresponding subformula f_q, the *contexts* of f_q is a set of variables defined as follows. Let $(q_1, \ldots, q_d)$ denote the vertices in T_{td} on the path from the root to q (excluding q itself). Then the contexts of f_q is the set $Contexts(f_q) = S(q_1) \cup S(q_2) \cup \ldots \cup S(q_d)$. Intuitively, the context of f_q is the set of all variables that are queried at nodes that lie along the path to q. Note that when we reach level $l = (q, 1)$ in DT, where the first variable of $S(q)$ is queried, we have already queried many variables, including all the variables in $Contexts(f_q)$. Thus the set of all variables queried up to level $l = (q, 1)$ can be partitioned into two groups relative to f_q: the irrelevant variables, and the set $Contexts(f_q)$ of relevant variables. We claim that at any arbitrary level $l = (q, r)$ in DT, the only nodes at level l that are actually traversed are those nodes $((q, r), \rho)$ where all irrelevant variables in ρ (with respect to f_q) are set to 0. The total number of such nodes at level $l = (q, r)$ is at most $2^{|Contexts(f_q)|+r}$ which is at most $2^{w \log n}$. Since this will be true for all levels, the total number of nodes in DT that are traversed is bounded by $n2^{w \log n}$. Thus, all that remains is to prove our claim.

Consider some node $s = ((q, r), \alpha)$ in DT. That is, $\alpha = \alpha^1 \alpha^2 \ldots \alpha^{q-1} b_1 \ldots b_{r-1}$, where for each i, α^i is an assignment to the variables in $S(i)$, and $b_1 \ldots b_{r-1}$ is an assignment to the first $r - 1$ variables in $S(q)$. Let the contexts of f_q be $S(q_1) \cup \ldots \cup S(q_d)$, $d \leq \log n$. Now suppose that α assigns a 1 to some non-context (irrelevant) variable, and say the first such assignment occurs at α^u_t, the t^{th} variable in α^u, $u \leq q - 1$. We want to show that the algorithm never traverses s.

Associated with α is a partial path in DT; we will also call this partial path α. Consider the subpath/subassignment p of α up to and including $\alpha^u_t = 1$. If α is traversed, then we start by traversing p. Since the last bit of p is 1 (i.e., $\alpha^u_t = 1$) when we get to this point, we have stored in the cache $ToComponents(f|_\rho)$ where ρ is exactly like p except that the last bit, α^u_t, is zero. Let j be the first node in $q_1, q_2, \ldots q_d$ with the property that the set of variables $S(j)$ are not queried in p. (On the path to q in T_{td}, j is the first node along this path such that the variables in $S(j)$ are not queried in p.) Then $ToComponents(f|_\rho)$ consists of three parts: (a) $ToComponents(f^L_j|_\rho)$, (b) $ToComponents(f_j|_\rho)$, and (c) $ToComponents(f^R_j|_\rho)$.

Now consider the path p' that extends p on the way to s in DT, where p' is the shortest subpath of α where all of the variables $S(i)$ for $i < j$ have been queried. The restriction corresponding to p' is a refinement of p where all variables in $S(1) \cup S(2) \cup \ldots S(j-1)$ are set. Since we have already set everything that occurs before j, we will only go beyond p' if some component of $ToComponents(f|_{p'})$ is not already in the cache. $ToComponents(f|_{p'})$ consists of three parts: (a) $ToComponents(f^L_j|_{p'})$, (b) $ToComponents(f_j|_{p'})$, and (c) $ToComponents(f^R_j|_{p'})$. Because we have set everything that occurs before j, all formulas in (a) will be known. Since p' and ρ agree on all variables that are relevant to f_j, $ToComponents(f_j|_{p'}) = ToComponents(f_j|_\rho)$ and hence these formulas in (b) in the cache. Similarly all formulas in (c) are in the cache since $ToComponents(f^R_j|_{p'}) = ToComponents(f^R_j|_\rho)$. Thus all components of $ToComponents(f|_{p'})$ are in the cache, and hence we have shown that we never traverse beyond p' and hence never traverse s. Therefore the total number of nodes traversed at any level $l = (q, r)$ is at most 2^{wd}, where d is the depth of q in T_{td}, as desired.

■

4 The connection between BAYES and #SAT

Here we formalize the problem BAYES and show how both BAYES and #SAT are instances of a more general problem. We also demonstrate how the standard algorithms for BAYES, as well as our new DPLL-based

algorithms for #SAT, can in fact be applied to this more general problem. This automatically allows us to apply any of these algorithms to either problem.

BAYES is concerned with the problem of computing probabilities in a Bayesian Network (BN). Invented by Pearl [Pea88], a Bayesian network is a triple $(\mathcal{V}, E, P)$ where $(\mathcal{V}, E)$ describes a directed acyclic graph (DAG), in which the nodes $\mathcal{V} = \{X_1, \ldots, X_n\}$ represent discrete random variables, edges represent direct correlations between the variables, and associated with each random variable X_i is a conditional probability table CPT (or function), $f_i(X_i, \pi(X_i)) \in P$, that specifies the conditional distribution of X_i given different assignments of values to its parents in $(\mathcal{V}, E)$, $\pi(X_i)$. A BN represents a joint distribution over the random variables $\mathcal{V}$ in which the probability of any assignment $(x_1, \ldots, x_n)$ to the variables is given by the equation

$$Pr(x_1, \ldots, x_n) = \prod_{i=1}^{n} f_i(x_i, \pi(x_i)),$$

where $f_i(x_i, \pi(x_i))$ is the CPT f_i evaluated at this particular assignment.

Thus a BN represents a particular distribution in a convenient and often compact way. Fundamental to the BN representation is that it supports a deep connection between the graphical structure and various independencies that hold of the distribution. We say that two random variables Z and Y are *independent* given a set of variables X if $P(z|x, y) = P(z|x)$ for all values x, y and z of the variables X, Y, and Z. In the distribution represented by a BN, a variable is independent of its nondescendants in the DAG given a set of values for all of its parents. Further independence statements that follow from these local statements can be computed in polynomial-time using the graph-theoretic notion of d-separation [Pea88].

At its most basic, BAYES is the problem of computing the distribution of a variable X_i given a particular assignment to some of the other variables α: i.e., $Pr(X_i|\alpha)$. Since X_i has only a finite set of values (say k values), this problem can be further reduced to that of computing the k values $Pr(X_i = j \wedge \alpha)$ and then normalizing them so that they sum to 1.

4.1 A more general problem

An important conceptual contribution of [Dec96] was to define a more general (and in our view more fundamental) problem, and to observe that BAYES is an instances of this problem. It is easy to show, as we do below, that #SAT is also an instance of this more general problem. The general problem is specified as follows. The input is $f = (V, F)$, where $F = \{f_1, \ldots, f_m\}$ is a set of functions and $V = \{x_1, \ldots x_n\}$ is a set of discrete valued variables. The range of each function is fixed depending on the problem, and can be boolean, or a subset of the integers or of the real numbers. Each function f_i has a domain set $e_i \subset V$. The problem is to compute $\sum_{x_1} \sum_{x_2} \cdots \sum_{x_n} f_1(e_1) \times f_2(e_2) \times \cdots \times f_m(e_m)$.

To solve an instance of BAYES, i.e., to compute $Pr(X_i = j \wedge \alpha)$ given a BN with some set of CPT functions F, we simply reduce the functions $f_i \in F$ by setting the variables assigned in α and setting X_i to the value j, and then sum out the remaining variables. This clearly is an instance of the general problem.

#SAT can equally well be recast as an instance, f, of this general problem. Let ϕ be a CNF formula over the variables $V = \{x_1, \ldots, x_n\}$. Let $f = (V, F)$ where each clause C_i of ϕ will be viewed as a function $f_i \in F$ on the underlying variables $variables(C_i)$. For each assignment to $variables(C_i)$, f_i will return 1 if the assignment satisfies C_i and 0 otherwise. Clearly the number of satisfying assignments for ϕ is $\sum_{x_1} \sum_{x_2} \cdots \sum_{x_n} f_1(variables(C_1)) \times f_2(variables(C_2)) \times \cdots \times f_m(variables(C_m))$:

Our DPLL algorithms described in Section 3 can be easily modified to solve the general problem. Now the input, f, to the DPLL algorithm is a set of functions, $F = \{f_1, \ldots, f_m\}$, over a set of discrete valued variables, $V = \{x_1, \ldots, x_n\}$, as described above. We want to compute $\sum_{x_1} \cdots \sum_{x_n} f_1(e_1) \times \cdots \times f_m(e_m)$. DPLL chooses a variable, x_i, and for each value a of x_i it recursively solves the reduced problem $f|_{x_i=a}$.

(Hence, instead of a binary decision tree it builds a k-ary tree). The reduced problem $f|_{x_i=a}$ is to compute

$$\sum_{x_1} \cdots \sum_{x_{i-1}} \sum_{x_{i+1}} \cdots \sum_{x_m} f_1(e_1)|_{x_i=a} \times \cdots \times f_m(e_m)|_{x_i=a},$$

where for an arbitrary c, $f_c(e_c)|_{x_i=a}$ is f_c reduced by setting $x_i = a$. That is, the problem is to sum out the remaining variables given that x_i has been set to a. #DPLL simply recurses until all the functions in F have been reduced to constant values (or one has been reduced to zero). #DPLLSimpleCache caches the reduced problem to avoid recomputing it, and #DPLLCache caches the solution to components of the reduced problem. Each component consists of a subset of the functions in the reduced problem, such that the union of the domain sets of these functions is disjoint from the domain set of all functions in any other component.

5 Comparison with known algorithms

5.1 Variable Elimination

Variable or bucket elimination (VE) [Dec96] is perhaps the most fundamental algorithm for BAYES. It has been shown to be closely related the all of the other algorithms [KDLC01] for BAYES, and it gives the same runtime and space guarantees as the best known BAYES algorithms.[4]

Now we describe the VE algorithm applied to the more general problem. Let $\mathcal{H}$ be the underlying hypergraph associated with the functions f_i. That is, there is one edge e_i corresponding to the domain set of each function f_i. Variable elimination begins by choosing an elimination ordering for the variables $V = \{x_1, \ldots, x_n\}$, call it $v_1^\pi, \ldots, v_n^\pi$. The algorithm proceeds in n phases. For the first phase, all functions involving v_1^π, $F_{v_1^\pi}$, are collected together, and a new function is obtained by "summing out" v_1^π. The new function takes the product of all functions in $F_{v_1^\pi}$ when v_1^π is set to one, and adds this to the product of the functions in $F_{v_1^\pi}$ when v_1^π is set to zero (or more generally it sums $F_{v_1^\pi}$ over all of v_1^π's possible values). This induces a new hypergraph, $\mathcal{H}_1$, where the set of functions $F_{v_1^\pi}$ is replaced by the single new function. This process continues until all n variables are "summed out". It is not too hard to see that the time and space complexity of this algorithm is exponential in the elimination width of $\mathcal{H}$ under π, and therefore the overall time is also bounded by the elimination width of $\mathcal{H}$.

Theorem 8 *The runtime and space of variable elimination for f is at most $nO(2^w)$ where w is the elimination width of the associated hypergraph $\mathcal{H}$.*

Since SAT is a special case of counting satisfying assignments, it is interesting to think about variable elimination applied in this more restrictive setting. We can solve SAT using the same formulation as that given above for #SAT. However, for SAT, rather than preserve the exact number of assignments. the new functions computed at each stage need only preserve whether or not the conjunction of the old functions is satisfiable. This can be accomplished by representing these new functions symbolically as a set of clauses. Furthermore, at the i-th stage, a function that preserves satisfiability can be obtained from the previous functions by generating all clauses that can be obtained by resolving on v_i^π, and then discarding all old clauses containing v_i^π. This resolution step corresponds to the summing out operation, and it is precisely the Davis-Putnam (DP) algorithm for satisfiability![5]

[4] In practice the join-tree clustering algorithm [LS88] is the most popular algorithm for BAYES. [KDLC01] shows how this algorithm can be reduced to a version of VE that remembers some of its intermediate results.

[5] Dechter and Rish [RD00] have previously made a connection between DP and variable elimination. They were thus able to show, as does the above, that DP runs in time $nO(2^w)$.

In some cases this symbolic representation of the functions can be substantially more compact than a straight-forward tabular representation. A useful example to study is applying variable elimination to a 2-CNF formula ϕ. If we merely want to determine whether or not ϕ is satisfiable, then all new functions generated by applying variable elimination (DP) to ϕ will have a 2-CNF representation, and thus the whole algorithm will run in polynomial-time. However, when we apply variable elimination to ϕ to count the number of satisfying assignments, it is not clear how to represent the new functions generated more succinctly than in a tabular (truth-table) form, and thus the runtime becomes exponential in the branch width.

5.2 Conditioning

Recursive conditioning [Dar01] is another algorithm for BAYES. It operates differently from VE, but it also solves the more general problem. Hence it also can be used to solve #SAT. As above, let ϕ be a CNF formula and let $\mathcal{H}$ be its associated hypergraph. Recursive conditioning begins with a branch decomposition T of $\mathcal{H}$ with width w and height d. It operates as a d-phase divide-and-conquer algorithm as follows. Let $label(T)$ be the set of variables in the label of T's root node. The algorithm computes the number of satisfying assignments for $\phi|_\alpha$ for all truth assignments α to the variables $label(T)$. The final solution is the sum of the solutions to these subproblems. Each subproblem is solved recursively, by setting variables as dictated by the branch decomposition.

The most efficient version of the algorithm remembers intermediate calculated values. The runtime and space of this version of the algorithm is $nO(2^w)$, where w is the branch width and n is the number of variables in ϕ.

Theorem 9 *[Dar01] The runtime and space of recursive conditioning is at most $nO(2^w)$ where w is the branch width of the hypergraph associated with the functions f_i of the problem.*

6 Advantages of DPLL

As noted in the introduction, DPLL tends to display an average case performance that is vastly superior to its worst case performance. In this section we point out a few reasons why our various versions of #DPLL might retain this kind of superiority in their average case behavior.

In section 4 we described how DPLL solves the input problem by solving a set of reduced instances of the problem where the reductions arise from assigning variables. Once we make some sequence of variable assignments in the DPLL tree, some of the original functions of the problem might be reduced to constants. For example, if every variable in e_i, the domain of one of the input functions f_i, has been set, then f_i will be reduced to a single value (its value when evaluated at this assignment). If any of the input functions reduce to zero, our DPLL algorithms need not descend any further—the reduced subproblem must have value 0.[6] In VE the corresponding situation is when one of the intermediate functions, g_i, produced by summing out some of the variables, has value 0 for some setting of its inputs. In VE there is no obvious way of gaining compuational efficiency from this: g_i is computed all at once after all of the previous functions have been computed. This kind of early termination supported by DPLL is the key reason for its superior average case behavior.

Dynamic variable ordering is another essential component in DPLL's efficiency. Our algorithms retain the ability to use dynamic variable orderings. In particular, say that the input problem is f, and we first branch on the assignments $x_i = 1$ and $x_i = 0$. There is no reason to suppose that a good elimination ordering for solving $f|_{x_i=1}$ is also good for solving $f|_{x_i=0}$. Our algorithms are free to choose different orderings for each subproblem (and recursively, different orderings for the subsequent decompositions of

[6] For #SAT this corresponds to the situation where a clause becomes empty.

these subproblems). VE is forced to use a single elimination order since it does not decompose the problem in this way. Recursive conditioning potentially has some flexibility, but it is still forced to use an ordering that separates the problem into disjoint pieces.

Finally, it can be that some of the input functions become constant prior to all of their variables being set (e.g., a clause might vanish because one of its literals has become true), or they might become independent of some of their remaining variables. This means the subproblems $f|_{x_i=1}$ and $f|_{x_i=0}$ might have quite different underlying hypergraphs. Our DPLL-based algorithms automatically take advantage of this fact, since they work on these reduced problems separately. VE, on the other hand, does not decompose the problem in this way, and hence cannot take advantage of this structure. In BAYES this situation corresponds to context-specific independence where random variable X might be dependent on the set of variables W, Y, Z when we consider all possible assignments to these variables (so $f(X, W, Y, Z)$ is one of the input functions), but when $W = \texttt{True}$ it might be that X becomes independent of Y (i.e., $f(X, W, Y, Z)|_{W=1}$ might be a function $g(X, Z)$ rather than $g(X, Y, Z)$). Currently only ad-hoc methods have been proposed [BFGK96] to take advantage of this kind of structure.

7 Conclusions

In this paper we have studied DPLL with caching and analyzed the performance of various types of caching for counting satisfying assignments and for Bayesian inference. The same caching methods can also be applied to the special case of SAT. Using the same ideas, it can be shown that DPLL with simple caching can polynomially simulate ordered and regular resolution. This is interesting since it is known that ordered and regular resolution provide exponential savings in some instances over DPLL, but it has been hard to take advantage of these savings without a clear procedure for implementing regular resolution. Our results are thus related to the recent paper [AR02] that basically show that ordered resolution proofs of satisfiability (a special type of regular resolution proof) can give rise to proofs that are substantially more efficient than the width-based algorithm.

We rephrase a problem posed by Rina Dechter and [AR02], namely is it possible to solve SAT or #SAT simultaneously in time $poly(n)2^{O(w)}$ and polynomial space? If not, can one give a hardness result along these lines?

The permanent and determinant of a boolean matrix are important algebraic functions that fit very naturally into the general framework discused in Section 4. For both of these functions, the sum is over the set of all permutations, and thus the underlying dependency graph will be highly interconnected. It is a classic result that the decision problem corresponding to the permanent is #P complete whereas the determinant is computable in polynomial-time. Recent work establishes that approximating the permanent is also polynomial-time computable [JSV01]. Is it possible to show that there is an efficient DPLL style algorithm for these polynomial time problems?

8 Acknowledgements

We thank Michael Littman for many valuable conversations and insights related to this work.

References

[AR02] A. Aleknovich and A. Razborov. Satisfiability, branch-width and tseitin tautologies. In *Proceedings of FOCS*, 2002.

[BCC+99] A. Biere, A. Cimatti, E.M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using sat procedures instead of bdds. In *Proc. 36th Design Automation Conference*, pages 317–320. IEEE Computer Society, 1999.

[BFGK96] Craig Boutilier, Nir Friedman, Moises Goldszmidt, and Daphne Koller. Context-specific independence in bayesian networks. In *Proceedings of the Twelfth Annual Conference on Uncertainty in Artificial Intelligence (UAI 96)*, pages 115–123, Portland, OR, 1996.

[BL99] E. Birnbaum and E. L. Lozinskii. The good old Davis Putnam procedure helps counting models. *J. Artif. Intell. Research (JAIR)*, 10:457–477, 1999.

[BP96] Paul W. Beame and Toniann Pitassi. Simplified and improved resolution lower bounds. In *Proceedings 37th Annual Symposium on Foundations of Computer Science*, pages 274–282, Burlington, VT, October 1996. IEEE.

[BP00] R. J. Bayardo and J. D. Pehoushek. Counting models using connected components. In *Proceedings of the AAAI National Conference (AAAI)*, pages 157–162, 2000.

[Dar01] Adnan Darwiche. Recursive conditioning. *Artificial Intelligence*, 126:5–41, 2001.

[Dec96] Rina Dechter. Bucket elimination: A unifying framework for probabilistic inference. In *Proceedings of the 12th Conference on Uncertainty in Artificial Intelligence (UAI-96)*, pages 211–219. Morgan Kaufmann Publishers, 1996.

[DLL62] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 4:394–397, 1962.

[Dub91] Olivier Dubois. Counting the number of solutions for instances of satisfiabiltiy. *Theoretical Computer Science*, 81:49–64, 1991.

[Hak85] A. Haken. The intractability of resolution. *Theoretical Computer Science*, 39:297–305, 1985.

[Hec93] David Heckerman. Causal independence for knowledge acquisition and inference. In *Proceedings of the 9th Conference on Uncertainty in Artificial Intelligence (UAI93)*, pages 122–127, Washington, D.C., 1993.

[JSV01] M. Jerrum, A. Sinclair, and E. Vigoda. A polynomial-time approximation algorithm for the permanent of a matrix with non-negative entries. In *Proceedings of the 33rd ACM Symposium on Theory of Computing*, 2001.

[KDLC01] K. Kask, R. Dechter, J. Larrosa, and F. Cozman. Unifying tree-decomposition schemes for automated reasoning. Technical Report R92, UC Irvine, 2001.

[LMP00] Michael L. Littman, Stephen M. Majercik, and Toniann Pitassi. Stochastic Boolean satisfiability. *Journal of Automated Reasoning*, 2000. To appear.

[LPI01] M. Littman, T. Pitassi, and R. Impagliazzo. New and old algorithms for belief net inference and counting satisfying assignments. Unpublished manuscript, 2001.

[LS88] S.L. Lauritzen and D.J. Spiegelhalter. Local computation with probabilities on graphical structures and their application to expert systems. *Journal of the Royal Statistical Society Series B*, 50(2):157–224, 1988.

[MMZ+01] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient sat solver. In *Proc. of the Design Automation Conference (DAC)*, 2001.

[Pea88] J. Pearl. *Probabilistic Reasoning in Intelligent Systems*. Morgan Kaufmann, San Mateo, CA, 2nd edition, 1988.

[PP91] D. Poole and G. M. Provan. What is the most likely diagnosis? In P.P. Bonissone, M. Henrion, L.N. Kanal, and J.F. Lemmer, editors, *Uncertainty in Artificial Intelligence 6*, North Holland, 1991. Elsevier Science Publishers B.V.

[RD00] I. Rish and R. Dechter. Resolution versus search: Two strategies for SAT. *Journal of Automated Reasoning*, 24(1):225–275, January 2000.

[Rot96] Dan Roth. On the hardness of approximate reasoning. *Artificial Intelligence*, 82(1–2):273–302, 1996.

[RS91] Neil Robertson and P.D. Seymour. Graph minors x. obstructions to tree-decomposition. *Journal of Combinatorial Theory, Series B*, 52:153–190, 1991.

[Val79a] Leslie G. Valiant. The complexity of computing the permanent. *Theoretical Computer Science*, 8:189–201, 1979.

[Val79b] Leslie G. Valiant. The complexity of enumeration and reliability problems. *SIAM Journal of Computing*, 9:410–421, 1979.

[Zha96] Wenhui Zhang. Number of models and satisfiability of sets of clauses. *Theoretical Computer Science*, 155:277–288, 1996.

ECCC ISSN 1433-8092
http://www.eccc.uni-trier.de/eccc
ftp://ftp.eccc.uni-trier.de/pub/eccc
ftpmail@ftp.eccc.uni-trier.de, subject 'help eccc'