Algorithms and Complexity Results 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

Abstract

Bayesian inference is an important problem with numerous applications in probabilistic reasoning. Counting satisfying assignments is a closely related problem of fundamental theoretical importance. In this paper, we show that plain old DPLL equipped with memoization (an algorithm we call #DPLLCache) can solve both of these problems with time complexity that is at least as good as state-of-the-art exact algorithms, and that it can also achieve the best known time-space tradeoff. We then proceed to show that there are instances where #DPLLCache can achieve an exponential speedup over existing algorithms.

1 Introduction

Bayesian inference (BAYES) is an important and well-studied problem with numerous practical applications in probabilistic reasoning [16]. #SAT is also a well-studied problem that is of fundamental theoretical importance. These two problems are known to be closely related. In particular, the decision versions of both #SAT and BAYES are #P-complete [21, 22], and there are *natural* polynomial-time reductions from each problem to the other [14].

A more direct relationship between these two problems arises from the observation that they are both instances of a more general "sum of products" problem (SUMPROD).[1] Perhaps the most fundamental algorithm for SUMPROD (developed in a general way in [11]) is based on the idea of eliminating the variables of the problem one by one following some fixed order. This algorithm is called variable elimination (VE), and it is the core notion in almost all state-of-the-art exact algorithms for SUMPROD (and BAYES).

A simple observation is that the Davis-Putnam algorithm (DP) for satisfiability [10] is an instance of variable elimination. This observation is interesting because the alternate algorithm for satisfiability, the backtracking search DPLL algorithm [9], is known to be in practice *vastly superior* to DP. For example, in experimental data [20], a version of DP utilizing modern heuristics for computing good elimination orders performed much worse than DPLL based algorithms. Its behavior on the "jhn" family of problems was typical. This family contains 50 problems each with 100 variables, 34 of which are unsatisfiable. The fastest DPLL based procedure is able to solve *all* 50 problems in 0.86 CPU seconds. The variable elimination DP algorithm was unable to solve any of these problems, either running out of memory, or exceeding a 10,000 CPU seconds time bound on *each* problem.

This performance gap raises the question of whether or not DPLL like algorithms can be used to solve SUMPROD and BAYES. In this paper we provide some affirmative answers to this question. In particular, we present new and simple extensions of the DPLL procedure that allow it to achieve the same performance guarantees as state-of-the-art exact algorithms for BAYES, in terms of both time and space. We also present instances where our new DPLL based algorithms can achieve an exponential speedup over existing algorithms. Besides these theoretical results, there are also good reasons to believe that our DPLL based algorithms have the potential to perform much better than their worst case guarantees on problems that arise from real domains. In other work, [2], we have investigated in more depth the practical application of the ideas presented here to the problem of BAYES, with very promising results.

An outline of the paper follows. In Section 2, we define SUMPROD, #SAT, and BAYES, and review two core state-of-the-art exact algorithms for BAYES. In Section 3, we discuss DPLL-based algorithms with caching for solving #SAT and SUMPROD and prove that our algorithms achieve the best known time and space guarantees for these problems. In Section 4, we introduce a framework for comparing nondeterministic algorithms for these problems. This allows us to prove that #DPLLCache can efficiently simulate known exact algorithms, and can moreover achieve exponential speedup on some instances.

*This research funded by governments of Ontario and Canada through their NSERC and PREA programs.

[1] Dechter [11] casts BAYES as an instance of the more general problem SUMPROD. That #SAT is also an instance of SUMPROD is probably known, but we have not seen it stated explicitly in the literature.

2 Background

SUMPROD: In [11] it was show that BAYES and many other problems are instances of a more general problem that we will call SUMPROD (sum-of-products). The input to SUMPROD is a pair $(\mathcal{V}, \mathcal{F})$, where $\mathcal{F} = \{f_1, \ldots, f_m\}$ is a set of functions and $\mathcal{V} = \{X_1, \ldots, X_n\}$ is a set of discrete valued variables. The range of each function is fixed depending on the problem (typically boolean or the reals). Each function f_i has a *domain set* $E_i \subset \mathcal{V}$. The problem is to compute $\sum_{X_1} \sum_{X_2} \cdots \sum_{X_n} \prod_{i=1}^{m} f_i(E_i)$: i.e., the sum over all assignments of values to the variables $\mathcal{V}$ of the product of the f_i evaluated at those assignments.

BAYES: BAYES is the problem of computing probabilities in a Bayesian Network (BN). Developed by Pearl [16], a Bayesian network is a triple $(\mathcal{V}, E, \mathcal{P})$ where $(\mathcal{V}, E)$ describes a directed acyclic graph, 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), $f_i(X_i, \pi(X_i)) \in \mathcal{P}$, that specifies the conditional distribution of X_i given assignments of values to its parents $\pi(X_i)$ in $(\mathcal{V}, E)$. 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 f_i evaluated at this particular assignment.

The generic BAYES problem is to compute the posterior 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 k values, this problem can be further reduced to that of computing the k values $Pr(X_i = d_j \wedge \alpha)$ and then normalizing them so that they sum to 1. $Pr(X_i = d_j \wedge \alpha)$ can be computed by making all of the assignments in α as well as $X_i = d_j$, and then summing out the other variables from the joint distribution $Pr(x_1, \ldots, x_n)$. Given the above product decomposition of $Pr(x_1, \ldots, x_n)$, this is equivalent to reducing the functions $f_i \in \mathcal{P}$ by setting the variables assigned in α and $X_i = d_j$, and then summing their product over the remaining variables; i.e., it is an instance of SUMPROD.

SAT and #SAT: Let $\mathcal{V} = \langle X_1, X_2, \ldots, X_n \rangle$ be a collection of n Boolean variables, and let $\phi(\mathcal{V})$ be a k-CNF Boolean formula on these variables with m clauses. **SAT** asks, given a Boolean formula $\phi(\mathcal{V})$ in k-CNF, does it have a satisfying assignment? The **#SAT** search problem asks, given a Boolean formula $\phi(\mathcal{V})$ in k-CNF, how many of its assignments are satisfying? #SAT can be expressed as an instance of SUMPROD as follows. We view each clause C_i of ϕ as being a function f_i on the variables in C_i, $Vars(C_i)$. f_i evaluates to 1 on a particular assignment to these variables iff the assignment satisfies C_i. Let $(\mathcal{V}, \mathcal{F})$ be the instance of SUMPROD where $\mathcal{F}$ is the set of "clause" functions f_i. Clearly the number of satisfying assignments for ϕ is $\sum_{X_1} \sum_{X_2} \cdots \sum_{X_n} \prod_{i=1}^{m} f_i(Vars(C_i))$.

Exact Algorithms for SUMPROD: Next we briefly review two prominent exact algorithms for BAYES. Both of these algorithms in fact solve the more general problem SUMPROD and we present them in that general context. These algorithms are in fact nondeterministic algorithms that should be considered to be families of procedures, each member of which is a particular deterministic realization.

Variable Elimination: The most fundamental algorithm for BAYES is variable or bucket elimination (VE) [11].[2] Given an instance $(\mathcal{V}, \mathcal{F})$ of SUMPROD, we define its **underlying hypergraph** $\mathcal{H}$. The vertices of $\mathcal{H}$ are the variables $\mathcal{V}$, and its hyperedges are the domain sets E_i of the functions f_i. Variable elimination begins by choosing an elimination ordering, π for the variables $\mathcal{V} = \{X_1, \ldots, X_n\}$: $X_{\pi(1)}, \ldots, X_{\pi(n)}$. (This is the nondeterministic part of the computation.) The algorithm then proceeds in n phases. In the first phase, all functions involving $X_{\pi(1)}$, $\mathcal{F}_{X_{\pi(1)}}$, are collected together, and a new function, F_1 is computed by "summing out" $X_{\pi(1)}$. The new function sums the product of all the functions in $\mathcal{F}_{X_{\pi(1)}}$ over all of $X_{\pi(1)}$'s values. Specifically, for any assignment α to the other variables in $\mathcal{F}_{X_{\pi(1)}}$, we have that $F_1(\alpha) = \sum_{d \in \text{vals}(X_{\pi(1)})} \prod_{f \in \mathcal{F}_{X_{\pi(1)}}} f(\alpha, X_{\pi(1)} = d)$. This induces a new hypergraph, $\mathcal{H}_1$, where the hyperedges corresponding to the set of functions $\mathcal{F}_{X_{\pi(1)}}$ are replaced by a single hyperedge corresponding to the new function F_1. The process then continues to sum out $X_{\pi(2)}$ from $\mathcal{H}_1$ and so on until all n variables are summed out.

Davis-Putnam is an instance of variable elimination. Consider applying variable elimination to the formulation of SAT given above. For SAT, the new functions F_i computed at each stage need only preserve whether or not the product of the functions in $\mathcal{F}_{X_{\pi(i)}}$ is 0 or 1, the exact number of satisfying assignments need not remembered. This can be accomplished by representing the F_i symbolically as a set of clauses. Furthermore, this set of clauses can be computed by generating all clauses that can be obtained by resolving on $X_{\pi(i)}$, and then discarding all old clauses containing $X_{\pi(i)}$. This resolution step corresponds to the summing out operation, and yields precisely the Davis-Putnam (DP) algorithm for satisfiability.[3]

Branch Decompositions: The second algorithm for SUMPROD requires the notion of a branch decomposition.

[2] In practice the junction-tree clustering (JT) algorithm [13] is the most popular algorithm for BAYES. This algorithm can be reduced to a version of VE that remembers some of its intermediate results and runs in the same time and space as VE.

[3] Dechter and Rish [17] have previously made a connection between DP and variable elimination. They were thus able to show, that DP runs in time $n2^{O(w)}$, where w is the branch width of the underlying hypergraph of the instance.

DEFINITION 1 (Robertson and Seymour [18]) 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 labels of leaves in the subtree rooted at n, and let B denote the union of the labels 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 *width* of T is the maximum size of any labelling in T. The **branch width** of $\mathcal{H}$ is the minimum of the widths of all possible branch decompositions of $\mathcal{H}$.

Recursive Conditioning (RC and RC$^+$): Recursive conditioning [8] (RC) is another algorithm for SUMPROD. Let $(\mathcal{V}, \mathcal{F})$ be an instance of SUMPROD and $\mathcal{H}$ be its underlying hypergraph. Recursive conditioning is a divide and conquer algorithm that instantiates the variables of $\mathcal{V}$ so as to break the problem into disjoint components. The original version of RC, as specified in [8], begins with a branch decomposition T of $\mathcal{H}$ with width w and height d. (This is the nondeterministic part of the computation.) T specifies a recursive decomposition of the problem used by RC as follows. Let $label(n)$ be the label of a node in T. Starting at r the root of T, RC computes $\mathcal{F}|_\alpha$ for all assignments α to the variables in $label(left(r)) \cap label(right(r))$, where $left(r)$ and $right(r)$ are the left and right children of r. Each such α renders the set of functions in the subtree below $left(r)$ (i.e., the leaf labels) disjoint from the functions below $right(r)$. By making $left(r)$ and $right(r)$ roots of their reduced subtrees, RC can then recursively sum out the product of the functions in those subtrees. Since these functions are disjoint, the sums can be multiplied to obtain the sum of the product of all of the functions below r conditioned on the instantiations in α. Repeating this process for each possible α and summing yields the final answer.

A simple extension of RC is to compute $\mathcal{F}|_\alpha$ iteratively rather than all at once. That is, rather than instantiate all of the variables in the set $\mathcal{L} = label(left(r)) \cap label(right(r))$ at once, these variables can be instantiated one at a time, and $\mathcal{F}$ reduced after each instantiation. We will call this extended version of RC, **RC**$^+$.

RC (RC$^+$) has the attractive feature that it can achieve a non-trivial space-time tradeoff, taking less time if it caches its recursively computed values. The DPLL based algorithms presented here share a number of features with RC; they also reduce and decompose the input problem by making instantiations, gain efficiency by caching, and achieve a similar space-time tradeoff. However, our algorithms are based on the paradigm of backtracking, rather than divide and conquer. As a result, they are not limited to following a static decomposition scheme specified by a fixed branch decomposition. As we will see, the limitation of a static decomposition scheme means that RC must perform exponentially worse than our algorithms on some instances.

Complexity and Branch Width: All known algorithms for BAYES and #SAT run in exponential-time in the worst case. However, when the branch width of the underlying hypergraph of the instance, w, is small, the above algorithms are much more efficient. In fact, both algorithms run in time and space $n^{O(1)}2^{O(w)}$.

The general problem of computing an optimal branch decomposition (i.e., one that has width equal to the branch width of $\mathcal{H}$) is NP-complete. However, Robertson and Seymour [19] give an algorithm for computing a branch decomposition with width that is within a factor of 2 of optimal and that runs in time $n^{O(1)}2^{O(w)}$, where w is the branch width of $\mathcal{H}$. By first running this *deterministic* algorithm to compute a good branch decomposition one obtains a deterministic version of RC that runs in time $2^{O(w \log n)}$ and space that is only linear in n. This deterministic version will run in time and space $n^{O(1)}2^{O(w)}$ if recursively computed values are cached. Similarly the decomposition can be used to obtain a deterministic version of VE that runs in time and space $n^{O(1)}2^{O(w)}$.

3 Using DPLL for #SAT and SUMPROD

We will first present all of our DPLL-based algorithms as algorithms for #SAT, and then later explain how to modify them to solve SUMPROD.

DPLL and #DPLL: DPLL is a nondeterministic algorithm for SAT, that has also been used to solve generalizations of SAT, including #SAT [3]. 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. It should be clear that DPLL is a nondeterministic procedure that generates a *decision tree* representing the underlying CNF formula. For solving SAT, the decision tree is traversed in a depth-first manner until either a satisfying path is encountered, or until the whole tree is traversed (and all paths falsify the formula).

A slight modification of DPLL allows it to count all satisfying assignments as it traverses the decision tree. Table 1 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 multiplying by 2^n, where n is the number of variables in ϕ.

Known exponential worst-case time bounds for DPLL also apply to #DPLL: for unsatisfiable formulas, both algorithms have to traverse an entire decision tree before terminating. This lower bound does not, however, help us discriminate between algorithms since *all* known algorithms for #SAT and BAYES take exponential-time in the worst-case. Nevertheless, it is not hard to see, however, that unlike VE and RC, #DPLL requires exponential time even on instances with small branch width. To see this, con-

Table 1 Standard DPLL algorithm for SAT — DPLL modified to count satisfying solutions

DPLL(ϕ)	#DPLL(ϕ)
if ϕ has no clauses, output "*satisfiable*" and HALT	if ϕ has no clauses, return 1
else-if ϕ does not contain an empty clause then	else-if ϕ has an empty clause, return 0
choose a variable x that appears in ϕ	else
Call DPLL($\phi|_{x=0}$)	Choose a variable x that appears in ϕ
Call DPLL($\phi|_{x=1}$)	return #DPLL($\phi|_{x=0}$) $\times \frac{1}{2}$ + #DPLL($\phi|_{x=1}$) $\times \frac{1}{2}$
return	

sider a 3CNF formula over $3n$ variables consisting of n disjoint clauses. This formula has branch width 3; however any complete decision tree has exponential size. Therefore #DPLL will require exponential time.

DPLL with caching: If one considers the above example of applying #DPLL to disjoint sets of clauses, it is clear 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. One way 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. The above form of caching, which we will call *simple caching* can be easily implemented as shown in Table 2.[4] 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.[5]

In addition to formulas stored in the cache there are also the following *obvious* formulas whose value is easy to compute. (1) The empty formula $\{\}$ containing no clauses has value 1. (2) Any formula containing the empty clause has value 0. 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. We can generalize to sets of formulas as follows. If Φ is a set of formulas we assign it a value equal to the product of the values of the formulas in it. 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 following (low complexity) subroutines are used to access the cache. (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 1 *For solving* #SAT *on n variables, there is an execution of #DPLLSimpleCache that runs in time bounded by $2^{O(w \log n)}$ where w is the underlying branch width of the instance. Furthermore, the algorithm can be made deterministic with the same time guarantees.*

Although the theorem shows that #DPLLSimpleCache does fairly well, its performance is not quite as good as the best BAYES algorithms (which run in time $n^{O(1)}2^{O(w)}$). One of our main contributions is to show that a variant of simple caching allows #DPLL to perform as well as the best known algorithms. We call the new algorithm #DPLLCache, and its implementation is given in Table 2.

The algorithm again creates a DPLL tree, 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 can later 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 can obtain the following upper bound on the runtime of #DPLLCache.

THEOREM 2 *For solving* #SAT *on n variables, there exists an execution of #DPLLCache that runs in time bounded by $n^{O(1)}2^{O(w)}$ where w is the underlying branch width of the instance. Furthermore, the algorithm can be made deterministic with the same time guarantees.*

Finally, there is a third variant of #DPLL with caching,

[4] Simple caching has been utilized before in [15], but without theoretical analysis.

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

Table 2 #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$\Big(\phi,$ GetValue$(\{\phi^-\}) \times \frac{1}{2}$ + GetValue$(\{\phi^+\}) \times \frac{1}{2}$ $\Big)$
  return

#DPLL algorithm with component caching

#DPLLCache(Φ)
If InCache(Φ), return
else
  $\Phi =$ 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$\Big(\phi,$ GetValue$(\Phi^-) \times \frac{1}{2}$ + GetValue$(\Phi^+) \times \frac{1}{2}$ $\Big)$
  return

#DPLLSpace, 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 algorithm 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. Specifically, #DPLLSpace is exactly like #DPLLCache, except that it calls RemoveFromCache($\Phi^- \cup \Phi^+$) just before returning.

THEOREM 3 *For solving* #SAT *on n variables, there is an execution of #DPLLSpace that 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. Furthermore, the algorithm can be made deterministic with the same time and space guarantees.*

We now prove these theorems. For the proof of theorems 1 and 2 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 [8], there is a branch decomposition of $\mathcal{H}$ of depth $O(\log m)$ and width $O(w)$. Also by [19], it is possible to find a branch decomposition, T_{bd}, such that T_{bd} has branch width $O(w)$ and depth $O(\log m)$, in time $poly(n)2^{O(w)}$. Thus our main goal for each of the three theorems will be to prove the stated time and space bounds for our DPLL-based procedures, when they are run on a static ordering that is easily obtainable from T_{bd}.

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 $Vars(f_i)$ be the set of variables in the (sub)formula f_i. Recall that in a branch decomposition the label of each vertex i, $label(i)$, is the set of variables in the intersection of $Vars(f_i)$ and $Vars(f - f_i)$. 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 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 $(i,1), \ldots, (i,j_i)$ denote the variables in $label(i)$ that do not appear in the label of an earlier 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: $\pi = \langle (i_1,1), (i_1,2), \ldots, (i_1,j_1), (i_2,1), \ldots, (i_2,j_2), \ldots, (i_s,1), \ldots, (i_s,j_s) \rangle$ $i_1 < i_2 < \ldots < i_s \leq z$, and $j_1, \ldots, j_s \leq w$. Note that for some vertices i of T_{bd}, nothing will be queried since all of the variables in its label may have occurred 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 $j_1 + j_2 + \ldots + j_s = n$ 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.

Proof of Theorem 1: We want to show that the size of the subtree of DT searched by #DPLLSimpleCache is at most $2^{O(w \log n)}$. When backtracking from 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 $j_1 + j_2 + \ldots + j_{q-1} + r$.

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} (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} $d \in O(\log m) = O(\log n)$.

Accounting for the time to search the cache, 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. Thus, #DPLLSimpleCache runs in time $(n2^{O(w \cdot \log n)})^2 = 2^{O(w \cdot \log n)}$. ■

Proof of Theorem 2: 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 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. And (2) no variable prior to x_l in the ordering π appears in an 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 (q must have a parent since the root has an empty label). 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 in the reduction of f_p whose form is determined solely by the settings of the variables in p and the r variables of q that have already been set. If this is the case, then there can be at most $2^{(w+r)} = 2^{O(w)}$ different components that x_l can appear in, and hence #DPLLCache can branch on x_l at most $2^{O(w)}$ times as each time one more of these components gets stored in the cache.

Now we prove the claim. The label of q consists of variables appearing in p's label and variables appearing in the label of q's sibling. Since all of the variables in $label(p)$ have been set, q and its sibling must now have an identical set of unqueried variables in their labels. Hence, q must be the left child of p as by the time the right child is visited in the ordering, x_l will have already been queried. Thus, at the time x_l is queried, f_p will have been affected only by the current setting of $label(p)$ (as these are the only variables it shares with the rest of the formula) and the first r queried variables from $label(q)$. That is, f_p can be in at most $2^{(w+r)}$ different configurations, and thus the component containing x_l can also be in at most this many different configurations.

Thus with n variables we obtain a bound on the number of branches in the decision tree explored by #DPLLCache of $n2^{O(w)}$. As in the proof of the previous theorem,

the overall runtime is at most quadratic in the number of branches traversed, to give the claimed bound of $n^{O(1)}2^{O(w)}$. ■

Proof of Theorem 3: For this proof, it will be more natural to work with a *tree decomposition* rather than a branch decomposition. Unlike branch width, tree width is defined over ordinary graphs. However, hypergraphs can reduced to ordinary graphs by replacing each hyperedge with a clique of edges.

DEFINITION 2 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 3 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 width of T is the maximum size of any labelling in T, and the **tree width** of G is the minimum of the widths of all possible tree decompositions of G.

Branch and tree width are essentially interchangeable.

LEMMA 4 (ROBERTSON AND SEYMOUR [18]) *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 $\mathcal{H}$.*

Let f be a k-CNF formula with n variables and m clauses and let $\mathcal{H}$ be the underlying hypergraph associated with f. We begin with a tree decomposition T_{td} of depth $O(\log m)$ and width $O(w)$ (computable in time $n^{O(1)}2^{O(w)}$). 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 the leaves of T_{td} to the left of f_i, and f_i^R, the conjunction of clauses of the leaves of T_{td} to the right of f_i. #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 $S(i) = (i, 1), \ldots, (i, j_i)$. If i is a non-leaf node with j and k being its left and right children, then the variables in $S(i)$ are exactly the variables that occur in both f_j and f_k 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: $S(1), S(2), \ldots, S(c)$, where some $S(i)$ may be empty. The underlying decision tree, DT, created by #DPLLSpace is a complete tree with n levels. As before 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 $s = ((q, r), \rho)$ (the r^{th} variable in $S(q)$).

#DPLLSpace 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 hits a node where all of the components of the formula to be computed are known, it can avoid traversing the subtree rooted at that node. Thus it searches 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 *context* 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 context of f_q is the set $Context(f_q) = S(q_1) \cup S(q_2) \cup \ldots \cup S(q_d)$. 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 $Context(f_q)$. 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 $Context(f_q)$ of relevant variables. We claim that at an 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 $2^{|Context(f_q)|+r}$ 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 context 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. This yields an overall runtime of $2^{O(w\log n)}$. ■

Using DPLL algorithms for BAYES: The DPLL algorithms described in this section can be easily modified to solve SUMPROD, and thus are able to solve BAYES directly. For SUMPROD, we want to compute $\sum_{X_1}\ldots\sum_{X_n}\prod_{j=1}^m f_j(E_j)$. DPLL chooses a variable, X_i, and for each value d of X_i it recursively solves the reduced problem $\mathcal{F}|_{X_i=d}$. (Hence, instead of a binary decision tree it builds a k-ary tree). The reduced problem $\mathcal{F}|_{X_i=d}$ is to compute $\sum_{X_1}\ldots\sum_{X_{i-1}}\sum_{X_{i+1}}\ldots\sum_{X_m}\prod_{j=1}^m f_j(E_j)|_{X_i=d}$, where $f_j(E_j)|_{X_i=d}$ is F_j reduced by setting $X_i = d$. #DPLLSimpleCache caches the reduced problem to avoid recomputing it, and #DPLLCache caches the solution to components of the reduced problem. It is not hard to show that the above three theorems continue to hold for #DPLL, #DPLLCache, and #DPLLSpace modified to solve SUMPROD.

4 A Framework for comparing BAYES and #SAT algorithms

The algorithms in the literature for BAYES as well as our new DPLL-based algorithms, are actually *nondeterministic* algorithms, or families of algorithms. In a seminal paper, Cook and Reckhow [7] defined *propositional proof systems* which give a way to classify families of algorithms for coNP-complete problems. In the same spirit, we can define propositional proof systems for *any* function, not just for the coNP-complete predicates, thus making it possible to compare different families of algorithms for BAYES and #SAT. Moreover, we extend the original Cook-Reckhow definition so that *time*, *space* and *nondeterministic bits* are explicit computational resources, rather than just *time*. This is motivated by theoretical as well as by practical considerations: real systems for BAYES can often run in time that is not polynomially bounded, but it is important that the space be kept nearly linear.

DEFINITION 4 Let f be a function from $\{0,1\}^n$ to $\mathcal{N}$. A *proof system* A for f is a uniform algorithm $V(x,y)$ where x is an instance of f and y is an additional binary advice string (or proof). V will be implemented by a 2-tape machine, where the input tape is read-once and the other work tape is unrestricted. (This detail is necessary in order to allow V to run in time that is linear in x.) Further, the following conditions are satisfied: (1) for all x,y pairs, $V(x,y)$ either outputs $f(x)$ or $V(x,y)$ outputs "nil"; (2) for all x there exists a y such that $V(x,y)$ outputs $f(x)$; (3) $V(x,y)$ runs in time bounded by $t_A(|x|,|y|)$ and space bounded by $s_A(|x|,|y|)$.

DEFINITION 5 Let f be a function from $\{0,1\}^n$ to $\mathcal{N}$. Let A and B be two proof systems for f as defined above, with time complexities t_A and t_B and space complexities s_A and s_B respectively. Then A *ptime-lspace simulates* B if there exists a function $\phi(x,y)$ such that: (1) $\phi(x,y)$ is computable by a deterministic machine that runs in output polynomial time, i.e., the runtime is a polynomial function of x and y and the output; (2) for all x,y, $t_A(|x|,|\phi(x,y)|)$ is polynomial in $t_B(|x|,|y|)$; (3) for all x,y, $s_A(|x|,|\phi(x,y)|)$ is linear in $s_B(|x|,|y|)$. We say that A *ptime simulates* B if conditions (1) and (2) above hold, but not necessarily condition (3).

Our definitions are consistent with the usual definitions for UNSAT. By decoupling the number of bits needed to

write down the proof and the verification time, we can for example, distinguish between the search spaces for ordered resolution versus resolution for UNSAT. In the former case, the proof is simply an ordering of variables and the verifier runs in time that is not necessarily polynomial in the size of the proof; in the latter case, the proof is the entire resolution proof but now the verifier is polynomial time.

5 Polynomial-simulation results

THEOREM 5 *RC ptime simulates VE.*

The proof of the above theorem is implicit in [8].

THEOREM 6 *#DPLLCache ptime simulates RC, RC$^+$ and VE. #DPLLSpace ptime-lspace simulates linear-space bounded RC.*

The idea behind the proof is as follows. RC when run on a particular branch decomposition can be simulated in polynomial time by #DPLLCache searching an ordered DPLL tree in which the variables are queried in the order given by a depth-first preorder traversal of the branch decomposition. A direct simulation can also be shown, where each execution step of RC with caching (RC) on (x, y) is simulated by #DPLLCache (#DPLLSpace) on (x, y'). Thus it can be shown that #DPLLCache, restricted to *static orderings* polynomially simulates RC with caching (#DPLLSpace polynomially simulates RC).

6 Lower bounds

THEOREM 7 *Neither RC, VE, nor RC$^+$ ptime simulates #DPLLCache. Moreover, neither RC, VE, nor RC$^+$ ptime simulates #DPLL.*

To prove this theorem we first observe that from a result of Johannsen [12], #DPLLCache and #DPLL can solve the negation of the propositional string-of-pearls principle [5] in time $n^{O(\log n)}$, when run with a *dynamic* variable ordering. This immediately gives the above result for VE and for RC, since the branch width of these problems is $O(n)$. However, the domination of RC on these problems is not so interesting, since as we pointed out in Section 2, there are some obvious ways to improve RC. The more substantial result, that RC$^+$, the improved version of RC, is also dominated on these problem, requires a non-trivial argument.

We continue to use the string-of-pearls principle, introduced in in [5]. From a bag of m pearls, which are colored red and blue, n pearls are chosen and placed on a string. The string-of-pearls principle says that if the first pearl in the string is red and the last one is blue, then there must be a red-blue or blue-red pair of pearls side-by-side somewhere on the string. The negation of the principle, $SP_{m,n}$, is expressed with variables $p_{i,j}$ and p_j for $i \in [n]$ and $j \in [m]$ where $p_{i,j}$ represents whether pearl j is mapped to vertex i on the string, and p_j represents whether pearl j is colored blue ($p_j = 0$) or red ($p_j = 1$). The clauses of $SP_{m,n}$ are as follows. (1) $\vee_{j=1}^{m} p_{i,j}$, $i \in [n]$. (2) $(\neg p_{i,j} \vee \neg p_{i,j'})$, $i \in [n]$ $j \in [m]$,$j' \in [m]$, $j \neq j'$. (3) $(\neg p_{i,j} \vee \neg p_{i',j})$, $i \in [n]$, $i' \in [n]$, $i \neq i'$, $j \in [m]$. (4) $(\neg p_{1,j} \vee p_j)$ and $(\neg p_{n,j} \vee \neg p_j)$, $j \in [m]$. (5) $(\neg p_{i,j} \vee \neg p_{i+1,j'} \vee \neg p_j \vee p_{j'})$, $1 \leq i < n$, $j \in [m]$, $j' \in [m]$, $j \neq j'$. (6) $(\neg p_{i,j} \vee \neg p_{i+1,j'} \vee p_j \vee \neg p_{j'})$, $1 \leq i < n$, $j \in [m]$, $j' \in [m]$, $j \neq j'$.

Johannsen [12] shows that $SP_{n,n}$ has quasipolynomial size tree Resolution proofs. It follows that #DPLLCache as well as #DPLL can also solve $SP_{n,n}$ in quasipolynomial time.

LEMMA 8 *$SP_{n,n}$ can be solved in time $n^{O(\log n)}$ by #DPLLCache and by #DPLL.*

THEOREM 9 *Let $\epsilon = 1/5$. Any VE or RC or ordered #DPLLCache algorithm for $SP_{n,n}$ requires time 2^{n^ϵ}.*

Proof: It suffices to prove that #DPLLCache under any static ordering requires time 2^{n^ϵ} for $SP_{m,n}$, $m = n$. By a static ordering, we mean that the variables are queried according to this ordering as long as they are mentioned in the current formula. That is, we allow a variable to be skipped over if it is irrelevant to the formula currently under consideration. We will visualize $SP_{n,n}$ as a bipartite graph, with n vertices on the left, and n pearls on the right. There is a pearl variable p_j corresponding to each of the n pearls, and an edge variable $p_{i,j}$ for every vertex-pearl pair. (Note that there are no variables corresponding to the vertices but we will still refer to them.)

Fix a particular total ordering of the underlying $n^2 + n$ variables, $\theta_1, \theta_2, \ldots, \theta_l$. For a pearl j, let $fanin_t(j)$ equal the number of edge variables $p_{k,j}$ incident with pearl j that are one of the first t variables queried. Similarly, for a vertex i, let $fanin_t(i)$ equal the number of edge variables $p_{i,k}$ incident with vertex i that are one of the first t variables queried. For a set of pearls S, let $fanin_t(S)$ equal the number of edge variables $p_{k,j}$ incident with some pearl $j \in S$ that are one of the first t variables queried. Similarly for a set of vertices S, $fanin_t(S)$ equals the number of edge variables $p_{i,k}$ incident with some vertex $i \in S$ that are one of the first t variables queried. Let $edges_t(j)$ and $edges_t(S)$ be defined similarly although now it is the set of such edges rather than the number of such edges. It should be clear from the context whether the domain objects are pearls or vertices.

We use a simple procedure, based on the particular ordering of the variables, for marking each pearl with either a **C** or with an **F** as follows. In this procedure, a pearl may at some point be marked with a **C** and then later overwritten with an **F**; however, once a pearl is marked with an **F**, it remains an **F** for the duration of the procedure. If a pearl j is marked with a **C** at some particular point in time, t, this means that at this point, the color of the pearl has already been queried, and $fanin_t(j)$ is less than n^δ, $\delta = 2/5$. If a pearl j is marked with an **F** at some particular point in time t, it means that at this point $fanin_t(j)$ is at least n^δ. (The

color of j may or may not have been queried.) If a pearl j is unmarked at time t, this means that its color has not yet been queried, and $fanin_t(j)$ is less than n^δ.

For l from 1 to n^2+n, we do the following. If the l^{th} variable queried is a pearl variable ($\theta_l = p_j$ for some j), and less than n^δ edges $p_{i,j}$ incident to j have been queried so far, then mark p_j with a **C**. Otherwise, if the l^{th} variable queried is an edge variable ($\theta_l = p_{i,j}$) and $fanin_l(j) \geq n^\delta$, then mark pearl j with an **F** (if not already marked with an **F**). Otherwise, leave pearl j unmarked.

Eventually every pearl will become marked **F**. Consider the first time t^* where we have either a lot of **C**'s, or a lot of **F**'s. More precisely, let t^* be the first time where either there are exactly n^ϵ **C**'s (and less than this many **F**'s) or where there are exactly n^ϵ **F**'s (and less than this many **C**'s.) If exactly n^ϵ **C**'s occurs first, then we will call this case (a). Extend t^* to t^*_a as follows. Let $\theta_{t^*+1}, \ldots, \theta_{t^*+c}$ be the largest segment of variables that are all pearl variables p_j such that j is already marked with an **F**. Then $t^*_a = t^* + c$. Notice that the query immediately following $\theta_{t^*_a}$ is either a pearl variable p_j that is currently unmarked, or an edge variable. On the other hand, if exactly n^ϵ **F**'s occurs first, then we will call this case (b). Again, extend t^* to t^*_b to ensure that the query immediately following $\theta_{t^*_b}$ is either a pearl variable p_j that is currently unmarked, or is an edge variable.

The intuition is that in case (a) (a lot of **C**'s), a lot of pearls are colored prematurely–that is, before we know what position they are mapped to–and hence a lot of queries must be asked. For case (b) (a lot of **F**'s), a lot of edge variables are queried thus again a lot of queries will be asked. We now proceed to prove this formally.

We begin with some notation and definitions. Let $f = SP_{n,n}$, and let $Vars(f)$ denote the set of all variables underlying f. A restriction ρ is a partial assignment of some of the variables underlying f to either 0 or to 1. If a variable x is unassigned by ρ, we denote this by $\rho(x) = *$. Let T be the DPLL tree based on the variable ordering θ. That is, T is a decision tree where variable θ_i is queried at level i of T. Recall that corresponding to each node v of T is a formula $f|_\rho$ where ρ is the restriction corresponding to the partial path from the root of T to v. The tree T is traversed by a depth-first search. For each vertex v with corresponding path p that is traversed, we check to see if $f|_p$ is already in the cache. If it is, then there is no need to traverse the subtree rooted below v. If it is not yet in the cache, then we traverse the left subtree of v, followed by the right subtree of v. After both subtrees have been traversed, we then pop back up to v, and store $f|_p$ in the cache. This induces an ordering on the vertices (and corresponding paths) of T that are traversed—whenever we pop back up to a vertex v (and thus, we can store its value in the cache), we put v (p) at the end of the current order.

LEMMA 10 *Let f be $SP_{n,n}$ and let π be a static ordering of the variables. Let ρ be a partial restriction of the variables. Then the runtime of #DPLLCache on (f, ρ) is not less than the runtime of #DPLLCache on $(f|_\rho, \pi')$, where π' is the ordering of the unassigned variables consistent with π.*

LEMMA 11 *For any restriction ρ, if $f|_\rho \neq 0$ and $\rho(p_{i,j}) = *$, then $p_{i,j}$ occurs in $f|_\rho$.*

Proof: Consider the clause $C_i = (p_{i,1} \vee \ldots \vee p_{i,m})$ in f. Since $p_{i,j}$ is in this clause, if $p_{i,j}$ does not occur in $f|_\rho$, then $C_i|_\rho$ must equal 1. Thus there exists $j' \neq j$ such that $\rho(p_{i,j'}) = 1$. But then the clause $(\neg p_{i,j} \vee \neg p_{i,j'})|_\rho = \neg p_{i,j}$ and thus $p_{i,j}$ does not disappear from $f|_\rho$. ■

COROLLARY 12 *Let θ be a total ordering of $Vars(f)$. Let ρ, ρ' be partial restrictions such that ρ sets exactly $\theta_1, \ldots, \theta_q$ and ρ' sets exactly $\theta_1, \ldots, \theta_{q'}$, $q' < q$. Suppose that there exists $\theta_k = p_{i,j}$ such that ρ sets θ_k but $\rho'(\theta_k) = *$. Then either $f|_\rho = 0$ or $f|_{\rho'} = 0$ or $f|_\rho \neq f|_{\rho'}$.*

Case (a). Let θ be a total ordering to $Vars(f)$ such that case (a) holds. Let P^C denote the set of exactly n^ϵ pearls that are marked **C** and let P^F denote the set of less than n^ϵ pearls (disjoint from P^C) that are marked **F**. Note that (the color of) all pearls in P^C have been queried by time t^*_a; the color of the pearls in P^F may be queried by time t^*_a, and the color of all pearls in $P - P^C - P^F$ have not been queried by time t^*_a. Note further that the total number of edges $p_{i,j}$ that have been queried is at most $n^{\epsilon+\delta} + n^{1+\epsilon} \leq 2n^{1+\epsilon}$.

We will define a partial restriction, M_a, to all but 2^{n^ϵ} of the variables in $\theta_1, \ldots, \theta_{t^*_a}$ as follows. For each $j \in P^F$, fix a one-to-one mapping from P^F to $[n]$ such that $range(j) \in edges_{t^*_a}(j)$ for each j. For each $j \in P^C$, for any variable $p_{i,j}$ queried in $\theta_1, \ldots \theta_{t^*_a}$, set $p_{i,j}$ to 0. For any vertex i such that all variables $p_{i,j}$ have been queried in $\theta_1, \ldots, \theta_{t^*_a}$, map i to exactly one pearl j such that $p_j \in P - P^C - P^F$. There are at most $2n^\epsilon$ such i. (This can be arbitrary as long as it is consistent with the one-to-one mapping already defined on P^F.) For all remaining $p_j \in P - P^C - P^F$ that have not yet been mapped to, set all queried variables $p_{i,j}$ to 0. For all pearls p_j in P^F that have been queried in $\theta_1, \ldots, \theta_{t^*_a}$, assign a fixed color to each such pearl (all Red or all Blue) so that the smallest Red/Blue gap is as large as possible. Note that the gap will be of size at least $n^{1-\epsilon}$. M_a sets all variables in $\theta_1, \ldots \theta_{t^*_a}$ except for the variables p_j, $j \in P^C$. Since there are n^ϵ such variables, the number of restrictions ρ to $\theta_1, \ldots, \theta_{t^*_a}$ consistent with M_a is exactly 2^{n^ϵ}. Let S denote this set of restrictions.

Let $f' = f|_{M_a}$ and let θ' be be the ordering on the unassigned variables consistent with θ. (The set of unassigned variables is: p_j, for $j \in P^C$, plus all variables in θ_k, $k > t^*_a$.) Let T' be the DPLL tree corresponding to θ' for solving f'. By Lemma 10, it suffices to show that #DPLLCache when run on inputs f' and T', takes time at least 2^{n^ϵ}.

Note that the first n^ϵ variables queried in T' are the pearl variables in P^C, and thus the set of all 2^{n^ϵ} paths of height exactly n^ϵ in T' correspond to the set S of all possible settings to these variables. We want to show that for each vertex v of height n^ϵ in T' (corresponding to each of the 2^{n^ϵ} settings of all variables in P^C), that v must be traversed by #DPLLCache, and thus the runtime is at least 2^{n^ϵ}.

Fix such a vertex v, and corresponding path $\rho \in S$. If v is not traversed, then there is some $\rho' \subseteq \rho$ and some σ such that σ occurs before ρ' in the ordering, and such that $f'|_\sigma = f'|_{\rho'}$. We want to show that this cannot happen. There are several cases to consider.

1a. Suppose that $|\sigma| \leq n^\epsilon$ and $\sigma \neq \rho'$. Then both ρ' and σ are partial assignments to some of the variables in P^C that are inconsistent with one another. It is easy to check that in this case, $f'|_{\rho'} \neq f'|_\sigma$.

2a. Suppose that $|\sigma| > n^\epsilon$, and the $(n^\epsilon+1)^{st}$ variable set by σ is an edge variable $p_{i,j}$. Because $|\rho'| \leq n^\epsilon$, $\rho'(p_{i,j}) = *$. By Corollary 12, it follows that $f'|_{\rho'} \neq f'|_\sigma$.

3a. Suppose that $|\sigma| > n^\epsilon$ and the $(n^\epsilon+1)^{st}$ variable set by σ is a pearl variable p_j. (Again, we know that p_j is unset by ρ'.) Since this is case (a), we can assume that $p_j \in P - P^C - P^F$. Call a vertex i *bad* if $P - P^F - P^C \subset edges_{t_a^*}(i)$. If i is bad, then $fanin_{t_a^*}(i)$ is greater than $n - 2n^\epsilon \geq n/2$. Since the total number of edges queried is at most $2n^{1+\epsilon}$, if follows that the number of bad vertices is at most $4n^\epsilon$. This implies that we can find a pair $i, i+1$ of vertices and a pearl j' such that: (1) $p_{i,j}$ is not queried in $\theta_1, \ldots, \theta_{t_a^*}$; (2) $p_{i+1,j'}$ is not queried in $\theta_1, \ldots, \theta_{t_a^*}$; (3) $p_{j'}$ is in $P - P^C - P^F$ and thus $p_{j'}$ is also not queried. Thus the clause $(\neg p_{i,j} \vee \neg p_j \vee \neg p_{i+1,j'} \vee p_{j'})|_{\rho'}$ does not disappear or shrink in $f'|_{\rho'}$, and thus $f'|_{\rho'} \neq f'|_\sigma$.

Case (b). Let θ be a total ordering to $Vars(f)$ such that case (b) holds. Now let P^C denote the set of less than n^ϵ pearls marked **C** and let P^F denote the set of exactly n^ϵ pearls marked **F**.

We define a partial restriction M_b to all but 2^{n^ϵ} of the variables in $\theta_1, \ldots, \theta_{t^*}$ as follows. Call a vertex i *full* if all variables $p_{i,j}$ have been queried in $\theta_1, \ldots, \theta_{t_b^*}$. There are at most n^ϵ full vertices. For each $j \in P^F$, we will fix a pair of vertices $F_j = (i_j, i'_j)$ in $[n]$. Let the union of all n^ϵ sets F_j be denoted by F. F has the following properties. (1) For each j, no element of F_j is full; (2) For each $j \in P^F$, $F_j \in edges_{t_b^*}(j)$; and (3) every two distinct elements in F are at least distance 4 apart. Since $fanin_{t_b^*}(j) \geq n^\delta$, and $\delta = 2/5 > \epsilon$, it is possible to find such sets F_j satisfying these criteria.

For each $p_{i,j}$ queried in $\theta_1, \ldots \theta_{t_b^*}$, where $j \in P^F$ and $i \notin F_j$, M_b will set $p_{i,j}$ to 0. For each $j \in P^C$, and for any variable $p_{i,j}$ queried in $\theta_1, \ldots \theta_{t_b^*}$, set $p_{i,j}$ to 0. For any full vertex i , map i to exactly one pearl j such that $p_j \in P - P^C - P^F$. (Again this can be arbitrary as long as it is consistent with a one-to-one mapping.) For the remaining $p_j \in P - P^C - P^F$ that have not yet been mapped to, set all queried variables $p_{i,j}$ to 0. For all pearls p_j in P^C, color them Red. For all pearls p_j in P^F that have been queried, assign a fixed color to each pearl.

The only variables that were queried in $\theta_1, \ldots \theta_{t_b^*}$ and that are not set by M_b are the edge variables, $p_{i,j}$, where $j \in P^F$, and $i \in F_j$. Let S denote the set of all 2^{n^ϵ} settings of these edge variables such that each $j \in P^F$ is mapped to exactly one element in F_j. Let $f' = f|_{M_b}$ and let T' be the DPLL tree corresponding to θ' for solving f', where θ' is the ordering on the unassigned variables consistent with θ. By Lemma 10, it suffices to show that #DPLLCache on f' and T' takes time at least 2^{n^ϵ}.

Note that the first $2n^\epsilon$ variables queried in T' are the variables $P_{i_j,j}$, $P_{i'_j,j}$, $j \in P^F$. The only nontrivial paths of height $2n^\epsilon$ in T' are those were each $j \in P^F$ is mapped to exactly one vertex in F_j, since otherwise the formula f' is set to 0. Thus, the nontrivial paths in T' of height $2n^\epsilon$ correspond to S. We want to show that for each such nontrivial vertex v of height $2n^\epsilon$ in T' (corresponding to each of the restrictions in S), that v must be traversed by #DPLLCache, and thus the runtime is at least 2^{n^ϵ}.

Fix a vertex v and corresponding path $\rho \in S$. Again we want to show that for any $\rho' \subseteq \rho$, and σ where σ occurs before ρ' in the ordering, that $f'|_{\rho'} \neq f'|_\sigma$. There are three cases to consider.

1b. Suppose that $|\sigma| \leq 2n^\epsilon$. If σ is nontrivial, then both ρ' and σ are partial mappings of the pearls j in P^F to F_j, that are inconsistent with one another. It is easy to check that in this case $f'|_\sigma \neq f'|_{\rho'}$.

2b. Suppose that $|\sigma| > 2n^\epsilon$ and the $(2n^\epsilon+1)^{st}$ variable set by σ is an edge variable $p_{i,j}$. Because $|\rho'| \leq 2n^\epsilon$, $\rho'(p_{i,j}) = *$. By Corollary 12, it follows that $f'|_\sigma \neq f'|_{\rho'}$.

3b. Suppose that $|\sigma| > 2n^\epsilon$ and the $(2n^\epsilon+1)^{st}$ variable set by σ is a pearl variable p_j. By the definition of t_b^*, we can assume that $p_j \in P - P^C - P^F$. By reasoning similar to case 3a, can find vertices $i, i+1$, and pearl $j' \in P - P^C - P^F$ such that none of the variable $p_{i,j}, p_{i+1,j}, p_{j'}$ are queried in $\theta_1, \ldots, \theta_{t_b^*}$. Thus the clause $(\neg p_{i,j} \vee \neg p_j \vee \neg p_{i+1,j'} \vee p_{j'})|_{\rho'}$ does not disappear to shrink in $f'|_{\rho'}1$, and therefore $f'|_{\rho'} \neq f'|_\sigma$.

Thus for each of the two cases, #DPLLCache on f' and T' takes time at least 2^{n^ϵ} and thus #DPLLCache on f and T takes time at least 2^{n^ϵ}. ■

7 Final Remarks

In this paper we have studied DPLL with caching, analyzing the performance of various types of caching for #SAT

and Bayesian inference. Similar caching methods have recently been explored for solving SAT [4], Our results also extend in a certain way the recent paper [1].

We have proved that from a theoretical point of view, #DPLLCache is just as efficient in terms of time and space as state-of-the-art exact algorithms for BAYES. Moreover, we have shown that on specific instances, #DPLLCache substantially outperforms other algorithms. It is an important question whether this advantage can be realized in practice, on typical real-world instances. Here we point out a few reasons why this might be the case.

In section 3 we described how our DPLL algorithms solve the input problem by recursively solving a set of reduced problems, where the reductions arise from assigning variables. These reduced problems might have structure that can be effectively exploited in the recursive invocations of DPLL. There are two prominent examples of this.

First, some of the subproblems might contain zero valued functions. In this case our algorithms need not recurse further—the reduced subproblem must have value 0.[6] In VE the corresponding situation is when one of the intermediate functions, F_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 computational efficiency from this: F_i is computed all at once.

Second, 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. For example, our algorithms are free to use dynamic variable orderings, where a different variable ordering is used solving each subproblem.

In BAYES this situation corresponds to context-specific independence where the random variable X might be dependent on the set of variables W, Y, Z when considering 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 $F(X, Z)$ rather than $F(X, Y, Z)$). Currently only ad-hoc methods have been proposed [6] to take advantage of this kind of structure.

8 Acknowledgements

We thank Stephen Cook and Michael Littman for valuable conversations.

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

References

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

[2] F. Bacchus, S. Dalmao, and T. Pitassi. Value elimination: Bayesian inference via backtracking search. In *Uncertainty in Artificial Intelligence (UAI-03)*, 2003.

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

[4] P. Beame, R. Impagliazzo, T. Pitassi, and N. Segerlind. Memoization and DPLL: Formula caching proof systems. Unpublished manuscript, 2003.

[5] M. Bonet, J. L. Esteban, N. Galesi, and J. Johannsen. Exponential separations between restricted resolution and cutting planes proof systems. In *FOCS*, pages 638–647, 1998.

[6] C. Boutilier, N. Friedman, M. Goldszmidt, and D. Koller. Context-specific independence in Bayesian networks. In *UAI 96*, pages 115–123, 1996.

[7] S. A. Cook and R. A. Reckhow. The relative efficiency of propositional proof systems. *Journal of Symbolic Logic*, 44(1):36–50, 1977.

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

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

[10] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7:201–215, 1960.

[11] R. Dechter. Bucket elimination: A unifying framework for reasoning. *Artificial Intelligence*, 113:41–85, 1999.

[12] J. Johannsen. Exponential incomparability of tree-like and ordered resolution. Unpublished manuscript, 2001.

[13] S. Lauritzen and D. 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.

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

[15] S. M. Majercik and M. L. Littman. Maxplan: A new approach to probabilistic planning. In *Artificial Intelligence Planning and Scheduling (AIPS)*, pages 86–93, 1998.

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

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

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

[19] N. Robertson and P. Seymour. Graph minors XIII. the disjoint paths problem. *Journal of Combinatorial Theory, Series B*, 63:65–110, 1995.

[20] L. Simon and P. Chatalic. Satex web site: http://www.lri.fr/~simon/satex/satex.php3.

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

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