Refined Bounds for Instance-Based Search Complexity of Counting and Other #P Problems

Lars Otten and Rina Dechter

Bren School of Information and Computer Sciences
University of California, Irvine, CA 92697-3425, U.S.A.
{lotten,dechter}@ics.uci.edu

Abstract. This paper presents measures for upper and lower bounding the instance-based complexity of AND/OR search algorithms for solution counting and related $\#P$ problems. This can be of utmost importance in selecting the right set of parameters for fitting an algorithm to a problem instance and in devising heuristics during execution. To this end we estimate the size of the search space, with special consideration given to the impact of determinism in a problem. The resulting schemes are evaluated empirically on a variety of problem instances; in many cases relatively tight bounds are obtained, far better than those implied by the tree width or hypertree width. Specific results are provided detailing how these measures can be useful for discriminating between variable orderings.

1 Introduction

This paper develops measures for upper and lower bounding the performance of search algorithms for solution counting in constraint networks and other related $\#P$ problems. It has been known for a while that the complexity of inference algorithms (e.g., join-tree clustering, variable elimination) is exponentially bounded by the tree width of the constraint problem's underlying graph structure. The base of the exponent is often taken to be the maximum domain size.

More accurate bounds were derived by looking at the respective domain sizes and their product in each cluster of a tree decomposition of the underlying graph [8]. These tighter bounds were used in selecting "good" variable orderings, for example. It was recently shown that these bounds are also applicable to search algorithms that explore the context-minimal AND/OR search graph [2].

The shortcoming of these bounds is that they are completely blind to context-sensitivity hidden in the relations and functions of the constraint network and especially determinism: When a problem possesses high levels of determinism, its tree width bound can be large while its search space can be extremely pruned, due to propagation and inconsistencies across relations.

Part of this shortcoming in worst-case complexity bounds is addressed by the more recent concept of hypertree decompositions [5]. It was shown that the maximum number of relations in the clusters of a hypertree decomposition (the hypertree width) exponentially bounds the problem complexity for constraint inference, a result that was extended to general graphical model inference in [7]. The base of the exponent in this case is the

relation tightness, thus allowing the notion of determinism to play a role. However, in practice this bound often turns out to be far worse than the tree width bound.

We recently introduced a more informed upper-bounding scheme, that selectively takes determinism into account by greedily covering variables with relations. We demonstrated its effectiveness empirically over a set of Bayesian networks and showed that the bounds it provides can in some cases be better by orders of magnitude [12]. These tighter bounds are desirable for a number of reasons:

1. We can better predict parameters of algorithms ahead of time (primarily the variable ordering for search), fitting the algorithm to the problem.
2. If we can dynamically adjust the estimates to reflect the current conditioning during search, it allows us to update parameters on the fly, e.g., for dynamic variable orderings.
3. In a distributed setup, search can often be implemented as centralized conditioning followed by independent solving of the conditioned subproblems on different machines. Better, adaptive bounds can help in balancing these two phases by means of selecting the central conditioning set dynamically and balancing distributed workloads [14].

In this paper we extend our earlier work in four ways. First, we refine the bounding scheme, which further improves the upper bounds we obtain: We "reuse" relations during the estimation process, projecting their scope down to the currently relevant variables and partially determining the impact of early deadends. This can also be viewed as a simple form of information propagation down the search graph. Secondly, we introduce a simple scheme for lower bounding, that uses a sampling-based SAT solution counting algorithm. Thirdly, we show that these schemes are applicable to constraint networks by presenting experiments on various sets of constraint problem instances. Finally, picking up on the issue of predicting algorithm parameters, we then investigate our bounds' ability to discriminate between different variable orderings and demonstrate that they are indeed informative in this respect. In general we obtain good results and in some cases our bounds can be shown to be very tight.

Section 2 gives definitions and some background. Section 3 motivates and describes our more refined bounding schemes. We present general empirical results in Section 4, while Section 5 focuses on the impact of different orderings on the computations. Section 6 concludes.

2 Preliminaries

We will assume a *graphical model*, given as a set of variables $X = \{x_1, \ldots, x_n\}$, their finite domains $D = \{D_1, \ldots, D_n\}$, a set of functions or relations $R = \{r_1, \ldots, r_m\}$, each of which is defined over a subset of X, and a combination operator (join, sum or product) over all functions. If we also have a marginalization operator such as $\min_X$ and $\max_X$ and obtain a *optimization problem*.

The special cases of reasoning tasks that we have in mind are constraint satisfaction problems (CSPs), using join as their combination operator. Given a CSP, we aim to find an assignment to all variables such that all constraints, typically expressed as relations,

are satisfied; alternatively we want to determine the the overall number of distinct solutions. An extension of CSPs are MAX-CSPs, where one wants to satisfy as many of the given constraints as possible, or weighted CSPs that have weights attached to violating constraints and call for minimizing the sum of these violation costs.

In the same way we can reason about Bayesian networks, where the primary tasks are belief updating and finding the most probable explanation. They are often specified using conditional probability functions defined on each variable and its parents in a given directed acyclic graph and use multiplication and summation or maximization as the combination and marginalization operators.

2.1 Problem Solving with Search

Two principal methods exist to solve CSPs and other reasoning problems: search (e.g., depth-first branch-and-bound, best-first search) and inference (e.g., variable elimination, join-tree clustering). Both can be shown to be time and space exponential in the problem instance's tree width w^* [1, 2, 7, 9], with a dominant factor of k^{w^*}, where k denotes the maximum domain size of the problem variables.

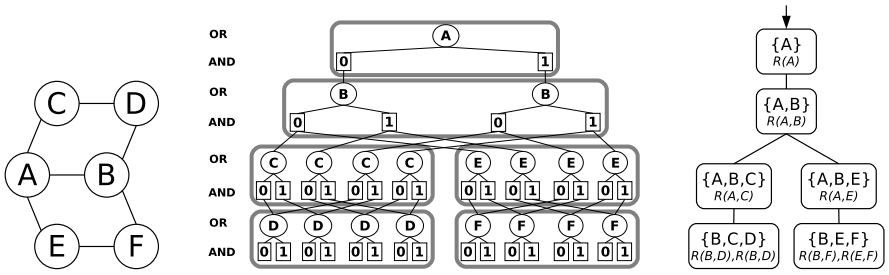
Search-based algorithms traverse the problem *search space*. Given a variable ordering d, the simplest way to perform search is to instantiate variables one at a time. This will define a search tree, where each node represents a state in the space of partial assignments. Leaf nodes signify either full solutions or dead ends. Standard depth-first algorithms typically have time complexity exponential in the number of variables and linear space complexity. If memory is available, one can apply caching to traversed nodes and retrieve their values when "similar" nodes are encountered, thereby trading space for time.

These traditional search spaces, however, don't fully capture the structure of the underlying graphical model. Introducing *AND* nodes into the search space can exploit independence of subproblems by effectively conditioning on values, thus avoiding some redundant computation. Since the size of the *AND/OR search tree* may be exponentially smaller than the traditional OR search one, any algorithm exploring the AND/OR space enjoys a better computational bound.

We can equally apply caching techniques to an algorithm exploring the AND/OR search tree. As a result this algorithm will effectively explore the *AND/OR search graph*. With caching, identical subproblems are recognized based on their context, which is a graphical model parameter that denotes the part of the search tree above that is relevant to the subproblem below [2].

Example 1. Assume a binary CSP with the primal graph in Figure 1(a) over variables $X = \{A, B, C, D, E, F\}$. We pick the ordering $d = A, B, C, D, E, F$. AND/OR search with caching will explore the search space depicted in Figure 1(b). We point out two things:

1. The AND nodes at the level of variable B have two children each. This denotes the fact that the problem decomposes at this point: After instantiating A and B, the subproblems over $\{C, D\}$ and $\{E, F\}$ are independent.



(a) (b) (c)

Fig. 1: Example primal graph, the AND/OR search graph along ordering $d = A, B, C, D, E, F$, and the corresponding bucket tree decomposition.

2. The OR nodes for D and F each have two ancestors. This "merge" is the effect of caching and symbolizes the fact that the subproblem is only dependent on some part of the search tree above – in this case the value of A is not relevant.

Obviously, the use of caching will avoid some redundant computations, thus reducing time complexity, at the cost of increased memory requirements. Assuming full caching, AND/OR search has been shown to exhibit both time and space complexity exponential in the problem's tree width [2].

2.2 Expressing Structure

If one wants to analyze the complexity of a given problem instance, it has proven useful to look at the underlying structure of interactions between variables. Given a constraint problem or graphical model in general, we will assume the usual definitions of the problem's *primal*, *dual*, and *hypergraph*.

It is well-known that a problem whose underlying graph exhibits tree structure can be solved efficiently [10, 13]. If this is not the case, we can aim to transform the problem into an equivalent one that exhibits tree structure [1, 7, 9]. Intuitively, we do this by clustering variables and the relations over them into groups, such that the set resulting clusters can be arranged into a tree:

Definition 1. *Let X, D, R be the variables, domains and relations of a constraint problem $\mathcal{P}$. A* **tree decomposition** *of $\mathcal{P}$ is a triple $\langle T, \chi, \psi \rangle$, where $T = (V, E)$ is a tree and χ and ψ are labeling functions that associate with each vertex $v \in V$ two sets, $\chi(v) \subseteq X$ and $\psi(v) \subseteq R$, that satisfy the following conditions:*

1. *For each $r_i \in R$, there is at least one vertex $v \in V$ such that $r_i \in \psi(v)$.*
2. *If $r_i \in \psi(v)$, then $scope(r_i) \subseteq \chi(v)$.*
3. *For each variable $x_i \in X$, the set $\{v \in V | x_i \in \chi(v)\}$ induces a connected subtree of T. This is also called the running intersection or the* connectedness property.

The **tree width** *of a tree decomposition $\langle T, \chi, \psi \rangle$ is $w = \max_v |\chi(v)| - 1$. The tree width w^* of $\mathcal{P}$ is the minimum tree width over all its tree decompositions.*

The problem of finding a tree decomposition of minimal tree width is known to be NP-complete. To obtain tree decompositions in practice, one can apply a triangulation algorithm to the problem's primal graph along an ordering and then construct the *bucket tree* by extracting and connecting the cliques for each variable, as described for instance in [13]. The ordering to use as the basis for the triangulation algorithm is often computed heuristically.

Example 2. Going back to Example 1, if we triangulate the problem's primal graph and extract each variable's *bucket* we obtain the tree decomposition shown in Figure 1(c), the *bucket tree decomposition*.

As it becomes evident already from Example 1, tree decompositions and AND/OR search (with caching) are very closely related; in particular, each cluster of the bucket tree decomposition corresponds to the layer of one variable in the search graph. In fact, it has been shown that in the absence of determinism both methods perform the same amount of work [11].

2.3 Problem Determinism

Many interesting practical problems, however, exhibit a significant degree of determinism. Search algorithms detect the resulting inconsistencies early in the search process and prune the respective portion of the search space. This can often lead to significant speedups, but it is not reflected in the standard, asymptotic worst-case bound $n \cdot k^w$ for the size of the AND/OR search graph. In order to take determinism into account for bounding, we define the following:

Definition 2. *The* **tightness** t_j *of a constraint or relation* r_j *of a CSP is the number of allowed tuples in the relation. (Similarly, for a conditional probability table of a Bayesian network, the tightness is the number of tuples with nonzero probability.) We denote with* $t = \max_j t_j$ *the maximum tightness of the relations in a problem instance.*

The motivation behind this notion is that we can store and process the relation r_j in a "compressed" form, with only the t_j revelavant tuples. Based on this we are able to exploit determinism through the concept of (generalized) hypertree decompositions, which was introduced for constraint networks in [5] and extended to general graphical models in [7]. As a subclass of tree decompositions, it was shown that it provides a stronger indicator of tractability than the tree width, while accounting for determinism to some extent.

Definition 3. *Let* $\mathcal{T} = \langle T, \chi, \psi \rangle$*, where* $T = (V, E)$ *be a tree decomposition of a constraint problem* $\mathcal{P}$ *with variables* X*, their domains* D *and relations* R*.* $\mathcal{T}$ *is a* **hypertree decomposition** *of* $\mathcal{P}$ *if the following additional condition is satisfied:*

4. *For each* $v \in V$*,* $\chi(v) \subseteq \bigcup_{r_j \in \psi(v)} scope(r_j)$ *.*

The **hypertree width** *of a hypertree decomposition is* $hw = \max_v |\psi(v)|$*. The hypertree width* hw^* *of* $\mathcal{P}$ *is the minimum hypertree width over all its hypertree decompositions.*

Similar to tree decompositions discussed above, finding a hypertree decomposition of minimal hypertree width is NP-complete in general, so one resorts to heuristic methods [3]. For a given decomposition it has been shown that the complexity of processing it to solve a constraint satisfaction problem is exponential in hw, with a dominant factor of t^{hw}, which accounts for determinism via the base t.

However, on practical problem instances the asymptotic bounds obtained from hypertree decompositions have been shown to be often far inferior to asymptotic bounds obtained from plain tree decompositions. Yet, the idea of using determinism when computing bounds remains promising. In the following we will therefore use this underlying principle – i.e., looking at the tightness of relations instead of the domain sizes of the problem variables – to derive better instance-based complexity bounds for search.

3 Search Space Estimation

Specifically, we will aim to estimate or bound the size of the AND/OR search graph explored by AND/OR search with caching, as depicted in Section 2.1.

Assume that a variable ordering $d = x_1, \ldots, x_n$ is fixed and that search instantiates variables first to last (while inference would proceed last to first). We note that the way the search space is decomposed by AND/OR search with caching can be represented by the bucket tree decomposition along the same ordering, because of the way it is constructed. For details we refer to [11], to illustrate we revisit our previous example:

Example 3. Consider the AND/OR search graph in Figure 1(b) and the bucket tree decomposition in Figure 1(c). It is easy to see that the decomposition clusters can be related to the "layers" of the search graph, i.e., the nodes associated with a variable and its values, as indicated within the search graph. For instance, the cluster $\{B, C, D\}$ represents the search layer for variable D and the fact that the subproblem only depends on B and C – and not A.

Based on this observation, we can equally partition the search space into "clusters" or "buckets", according to the corresponding bucket tree decomposition. Our approach of estimating the size of the entire search space will then be to estimate the portion of the search space in each cluster, subsequently summing over all clusters.

In the following we focus on a single cluster/layer of the search graph. To reiterate, this cluster will contain the search layer's variable, all of the already instantiated variables that are relevant to it, and zero or more relations over these variables.

We will discuss two methods, one that upper-bounds the number of search nodes, and one that computes a lower bound. Note that for simplicity we consider only the AND nodes of the search space, since OR nodes are actually not implemented as such in practice.

3.1 Upper-Bounding Cluster Size in Search

A straightforward upper bound is obtained by multiplying the domain sizes of all variables in the cluster [8]. If the set of clusters is given by $C = \{C_1, \ldots, C_n\}$ with cluster C_k containing variables $X_k \subseteq X$, we can formally define:

Algorithm *GreedyCovering*
Input: Set of variables $X = \{x_1, \ldots, x_r\}$ and set of relations $R = \{r_1, \ldots, r_s\}$, with x_i having domain size $|D_i|$ and r_j having tightness t_j
Output: A subset of R (that forms a partial covering of X)
Init: *Uncov* $:= X$, *Covering* $:= \emptyset$
(1) Find j^* that minimizes $q_j = t_j / \prod_{x_k \in I_j} |D_k|$, where $I_j =$ *Uncov* $\cap\, scope(r_j)$.
(2) If $q_{j^*} \geq 1$, terminate and return *Covering*.
(3) Add r_{j^*} to *Covering* and set *Uncov* $:=$ *Uncov* $\setminus\, scope(r_{j^*})$.
(4) If *Uncov* $= \emptyset$, terminate and return *Covering*.
(5) Goto (1).

Fig. 2: Greedy covering algorithm

$$twb := \sum_{k=1}^{n} \prod_{x_i \in X_k} |D_i| \,,$$

where $|D_i|$ is the domain size of variable x_i.

Note that this is closely related to the worst case complexity, since the tree width is just the maximum number of variables in any cluster of the decomposition minus 1. This, however, does not take determinism into account.

If all variables in a cluster are covered by the scopes of the relations in that cluster, we can take the product of the relations' tightnesses as an upper bound on the number of nodes, accounting for the maximum number of valid nodes – this is equivalent to the complexity analysis of processing hypertree decompositions from Section 2.3. If some variables are not covered, we multiply the tightness product by the domain sizes of the uncovered variables (thereby accounting for all possible combined instantiations).

However, since the scopes of the different relations in a cluster can overlap, this bound will typically be worse than the simple "product of domain sizes" bound twb. Consequentially, we use the latter as a starting point and exploit tight relations to improve upon it (this can be seen as combining the concept of tree and hypertree decompositions). In particular we want to find a (partial) covering of the variables by a subset of the relations which minimizes the bound we obtain, i.e. the product of the chosen relations' tightnesses multiplied by the domain sizes of the uncovered variables.

In essence this can be seen as a weighted, multiplicative variant of the well-known SET COVER problem, where one aims to cover a set or vertices by as few as possible subsets from a set of given subsets of the variables. The problem is generally NP-complete, but simple greedy approximations exist [6], which give rise to our method:

Initially we start with an empty covering (if we stop at this point, we get exactly the bound twb, i.e. the product of the variables' domain sizes). Then, for each relation r_j in the cluster, we compute the *coverage ratio* q_j as follows: Divide the relation's tightness t_j by the product over the domain sizes of the variables that have not yet been covered and are in the scope of r_j. We pick the relation for which the coverage ratio is the lowest and add it to the covering. We repeat this for as long as we can still find a relation with a coverage ratio less than 1. A formal description of the algorithm is given in Figure 2.

This will produce a set of relations as the covering, which might leave some variables uncovered. We can then multiply the tightness of the relations in the covering and

Algorithm *Compute-hwb*
Input: A bucket tree decomposition with clusters $C_1, \ldots, C_n$, where cluster C_k contains variables $X_k \subseteq X$ and relations $R_k \subseteq R$
Output: The bound hwb on the size of the search space
Init: $hwb := 0$
(1) for $i = 1$ to n:
(2) $R := R_k$.
(3) For every relevant relation r from the ancestral buckets, project it onto $scope(r) \cap X_k$ and add it to R with updated tightness t'_r.
(4) $Cov :=$ *GreedyCovering*(X_k, R).
(5) $hwb := hwb + \prod_{r_j \in Cov} t_j \cdot \prod_{x_i \in X_k \setminus Cov} |D_i|$.
(6) end for.
(7) Return hwb.

Fig. 3: Procedure to compute the overall upper bound hwb.

the domain sizes of the uncovered variables to obtain an upper bound on the number of nodes in the cluster [12].

Propagating Cluster Size Downwards. Computing this covering in each cluster independently does not take into account the propagation of determinism down the search tree, since the number of valid nodes in a cluster impacts the number of valid nodes in its child clusters. Fully incorporating this propagation is computationally expensive, but we can account for it mildly, as we describe next.

When assembling the covering in a cluster, we can consider any relation from the ancestral clusters higher up in the rooted tree decomposition (since their scope will have been fully instantiated at this point of the search). However, we can further improve upon this scheme: As we noted, during search only nodes representing valid assignments will be expanded and have their child nodes in the child cluster considered.

Therefore, when collecting the relations from all ancestral buckets as candidates for the covering of the current cluster, we project them down to the current clusters's scope. This will typically result in a relation with a smaller number of valid tuples (note that in practice we don't need to store the full relation but only the projected tightness). Since this projection will consider exactly those variables that a parent and child cluster share, it can be interpreted as propagation of information down the search tree. In practice, we found that for some problem instances this will decrease the bound by up to 30%.

Example 4. Assume we have a cluster containing 3 variables X, Y, Z, each with domain size 4, and 2 relations $r_1(X, Y)$ and $r_2(Y, Z)$ with tightness $t_1 = 9$ and $t_2 = 13$. Furthermore, assume we find that relation $r_3(W, X)$ with $t_3 = 10$ from the parent cluster is relevant. We project r_3 down to X and find the new tightness is $t'_3 = 3$. The worst case bound on the number of nodes is clearly $4^3 = 64$. In the first iteration of the *GreedyCovering* algorithm the coverage ratios of r_1, r_2, and r_3 will be computed as $q_1 = \frac{9}{16}$, $q_2 = \frac{13}{16}$, and $q_3 = \frac{3}{4}$, respectively. Therefore r_1 will be added to the covering, leaving only Z uncovered. The next coverage ratio of r_2 and r_3 will be computed as $\frac{13}{4}$ and $\frac{3}{1}$ (since the scope of r_3 is fully covered already). This is both greater than 1, therefore the algorithm terminates and r_2 and r_3 will not be part of the covering. The bound for this cluster will then be $t_1 \cdot |D_Z| = 9 \cdot 4 = 36$.

Proposition 1 (Overall Upper Bound). *Computing the bound for each cluster of the tree decomposition and summing up over the clusters yields an upper bound on the total number of nodes in the AND/OR search graph, which we denote* hwb.

The corresponding algorithm is given in Figure 3, its complexity is easily analyzed:

Theorem 1 (Complexity). *Given a constraint problem with* n *variables and* m *relations with maximal tightness* t. *If the bucket tree decomposition along a given variable ordering* d *has tree width* w, *the time complexity of* Compute-hwb *is* $\mathcal{O}(n \cdot m \cdot (t + w))$ *and the space complexity is* $\mathcal{O}(m + t)$.

Proof. Collecting and projecting (at the same time computing the tightness) of all relevant relations for a cluster takes time $\mathcal{O}(m \cdot t)$, the space needed for projecting relations and storing the resulting tightness values is $\mathcal{O}(t + m)$. The time complexity of the *GreedyCovering* procedure is linear both in the number of variables and the number of relations considered, i.e. it is worst-case bounded by $\mathcal{O}(w \cdot m)$; the procedure keeps track of the covering ratio for each relation, therefore space is $\mathcal{O}(m)$. Iterating over all of the n clusters produces the stated overall complexity bounds. □

3.2 Lower-Bounding Cluster Size

To obtain lower bounds on the the number of nodes in a cluster we employ a different scheme: In each cluster, we generate a SAT formula from all relevant relations (i.e., from within the current cluster and ancestral ones) and feed it to the sampling-based SAT solution counting algorithm SampleSearch-LB [4].

Internally, SampleSearch-LB first generates a set of biased solution samples from the backtrack-free distribution using the recently introduced sampling scheme SampleSearch and then corrects the bias using importance sampling to yield an unbiased estimate of the number of solutions. The Markov inequality is then used to convert the unbiased estimate to a lower bound on the solution counts with a required degree of confidence α. (Note that the bounds achieved are probabilistic in the sense that with probability $1 - \alpha$ they are not valid.) Empirical study in [4] shows that on many problems SampleSearch-LB yields tight lower bounds in practice with high probability.

Since we encode the invalid tuples of each relevant relation as the nogoods of the SAT formula, the number of solutions of the SAT problem will correspond to the number of valid nodes in the cluster.

Proposition 2 (Overall Lower Bound). *It is then easy to see that the sum of lower bounds on the solution counts in each cluster will be a lower bound on the size of the search graph explored by AND/OR search with caching; we call it* $satb$.

4 Experimental results

We ran a variety of empirical tests on a large set of different problem instances from various domains. For the most part, we experimented on problem instances from the repository of the CP'06 competition[1].

[1] http://www.cril.univ-artois.fr/∼lecoutre/research/benchmarks/benchmarks.html

For every instance, we report the number of variables n, the number of relations m, the maximum variable domain size k, and the maximum relation arity r. To quantify the degree of determinism in a problem, we report the median tightness ratio tr over all its relations, defined as the ratio of valid tuples in a (full) relation table (i.e., in a relation with a tightness ratio of 66%, two thirds of the tuples are valid).

We build a bucket tree decomposition of the problem along a minfill ordering and report the tree width w. We then compute and report our bounds twb, hwb, and $satb$. We also try to compute the exact size of the AND/OR search graph, denoted $\#cm$, by exploring it fully using a brute-force depth-first search scheme with no look-ahead or propagation. This can be seen as equivalent to the space explored by a brute-force counting algorithm. Note that computing this number is not always feasible due to limited computational resources (this case is denoted by "n/a" in the tables). To make comparing the values easier, we report the ratios $Q_t = \frac{twb}{\#cm}$, $Q_h = \frac{hwb}{\#cm}$, and $Q_s = \frac{satb}{\#cm}$.

Time. The time to compute twb and hwb is only a few milliseconds for all of the reported instances. To compute $satb$, the SAT sampler is ran with 1000 samples for each cluster: the runtime for the full problem instance on a 2.66GHz CPU ranges from several milliseconds for small instances to three seconds for the instance ssa-7552-038. We note again that the $\#cm$ value is directly proportional to the runtime of the exact algorithm; for example, on instance ssa-0432-003, running on a 2.66 GHz CPU, the brute-force algorithm generates around 43,000 nodes per second.

Results. Table 1 presents results on small *Dubois* instances (from the DIMACS problem set), two *Pret* instances that encode graph coloring problems, and several *SSA* problem instances derived from circuit fault analysis problems. Table 2 shows statistics on DIMACS *AIM* instances (3-SAT problems with 50 or 100 variables).

In Table 1 we also include a set of results that we obtained on randomly generated partial k-trees, which we adapted from Bayesian networks from the UAI'06 evaluation repository[2]. These instances had a certain degree of determinism enforced upon generating their tables and should therefore lend themselves nicely to our approach.

We also ran the bounding scheme on other instances that have a very low level of determinism (not reported here), and as one would expect twb and hwb as well as $\#cm$ are all very close to each other in these cases, if not the same.

4.1 Analysis of Results

Bound Improvements under Determinism. If we just want to analyze the effect of incorporating determinism into the bound, we can compare twb to hwb. Looking at Tables 1 and 2 we can see that the improvement is significant. For the small, less complex *Dubois* instances the decrease is around 40%, for the *Pret* instances it is roughly 28%. On the *SSA* problems the improvement ranges from close to 20%, for instance on instance ssa-7552-158, to over 60% on instance ssa-7552-038. On the *AIM* instances the bound decreases between 18% (instance aim-50-1-6-sat-1) to over 70% on aim-50-3-4-sat-4. The partial k-trees (*BN* instances) that have been created with enforced determin-

[2] http://ssli.ee.washington.edu/∼bilmes/uai06InferenceEvaluation/

instance	n	m	k	r	w	tr	twb	hwb	$satb$	$\#cm$	Q_t	Q_h	Q_s
dubois-20	60	40	2	3	3	0.50	774	466	95	458	1.69	1.02	0.21
dubois-21	63	42	2	3	3	0.50	814	490	99	482	1.69	1.02	0.21
dubois-22	66	44	2	3	3	0.50	854	518	108	510	1.67	1.02	0.21
dubois-23	69	46	2	3	3	0.50	894	542	111	534	1.67	1.01	0.21
dubois-24	72	48	2	3	3	0.50	934	566	115	558	1.67	1.01	0.21
dubois-25	75	50	2	3	3	0.50	974	586	115	578	1.69	1.01	0.20
dubois-26	78	52	2	3	3	0.50	1,014	610	119	301	1.68	1.01	0.20
dubois-27	81	54	2	3	3	0.50	1,054	634	123	626	1.68	1.01	0.20
dubois-28	84	56	2	3	3	0.50	1,094	658	128	650	1.68	1.01	0.20
dubois-29	87	58	2	3	3	0.50	1,134	686	135	678	1.67	1.01	0.20
dubois-30	90	60	2	3	3	0.50	1,174	710	140	702	1.67	1.01	0.20
dubois-50	150	100	2	3	3	0.50	1,974	1,190	220	1,182	1.67	1.01	0.19
dubois-100	300	200	2	3	3	0.50	3,974	2,390	420	2,382	1.67	1.00	0.18
pret-60	60	40	2	3	4	0.50	1,534	1,102	839	998	1.51	1.10	0.89
pret-150	150	100	2	3	4	0.50	3,934	2,862	2,303	2,598	1.54	1.10	0.84
ssa-0432-003	435	738	2	5	31	0.75	4,244,330	2,059,616	1,116,669	1,868,283	2.27	1.10	0.60
ssa-2670-130	1359	2366	2	5	31	0.75	160,631,566	123,388,312	104,689,598	106,638,207	1.51	1.16	0.98
ssa-7552-038	1501	2444	2	6	63	0.75	308,861,278	115,499,146	6,815,140	36,718,327	8.41	3.15	0.19
ssa-7552-158	1363	1985	2	5	31	0.50	90,702	74,406	56,863	69,365	1.31	1.07	0.82
ssa-7552-159	1363	1983	2	5	31	0.50	92,238	73,586	48,929	68,694	1.34	1.07	0.71
BN_105	40	44	2	21	18	0.62	2,477,054	363	69	131	18909	2.77	0.53
BN_107	40	46	2	21	21	0.62	29,983,742	1,643	191	272	110234	6.04	0.70
BN_109	40	46	2	20	20	0.62	13,054,974	4,052	1,309	2,531	5158	1.60	0.52
BN_111	40	45	2	20	19	0.62	8,406,270	2,299	465	979	8587	2.35	0.47
BN_113	40	47	2	21	21	0.62	18,916,350	2,752	336	630	30026	4.37	0.53

Table 1: Results for *Dubois*, *Pret*, *SSA*, and *BN* instances.

ism, are as expected very amenable to our approach and see an improvement of three to four orders of magnitude when going from twb to hwb by exploiting determinism.

Quality of hwb Bound. We now compare the upper bound hwb to the true size of the search space $\#cm$. On the small *Dubois* and *Pret* instances, the bound is pretty tight, getting within 1-2% and 10% of the true size, respectively. For the *SSA* instances the bound is around 10-15% off, with the exception of the ssa-7552-038 instance, where hwb is about three times the size of the true search space size.

Looking at the *AIM* instances, the picture is equally mixed: For the first, relatively simple instances with lower w, the bound hwb is within 10-20% of $\#cm$, but as the instances grow more complex and w increases, the bounds move farther away, to within 60-70% of the true value for the aim-50-2-0-sat class. For the most complex problem class aim-50-3-4-sat, the twb bound is even two orders of magnitude off.

On the partial k-trees BN_105-113, where our bound produced significant improvements over twb, hwb is not really close to $\#cm$ but, with six times the true size for BN_107 as the worst case, also not orders of magnitude off.

Quality of $satb$ Bound. The results for $satb$, however, are less impressive at this point. While we indeed obtain lower bounds in practice, their quality is mostly not quite satisfactory; for many instances $satb$ is 50% or more smaller than $\#cm$, in a few cases one order of magnitude. But even though these lower bounds are often inaccurate, they are still somewhat informative and can set the stage for future improvements. For example, on instance aim-100-1-6-sat-3 the interval $[satb, hwb]$ is definitely more informative then hwb alone.

instance	n	m	k	r	w	twb	hwb	$satb$	$\#cm$	Q_t	Q_h	Q_s
aim-50-1-6-sat-1	50	77	2	3	18	2,517,118	2,053,046	931,492	1,813,906	1.39	1.13	0.51
aim-50-1-6-sat-2	50	76	2	3	16	767,678	626,955	73,180	551,659	1.39	1.14	0.13
aim-50-1-6-sat-3	50	78	2	3	20	4,742,590	3,859,278	2,023,465	3,848,835	1.23	1.00	0.53
aim-50-1-6-sat-4	50	77	2	3	19	3,615,166	2,616,824	2,079,752	2,532,968	1.43	1.03	0.82
aim-50-1-6-unsat-1	50	69	2	3	15	377,502	256,482	26,806	211,168	1.79	1.21	0.13
aim-50-1-6-unsat-2	50	77	2	3	19	3,484,734	2,551,090	16,995	1,908,441	1.83	1.34	0.01
aim-50-1-6-unsat-3	50	70	2	3	17	1,190,910	971,254	43,382	685,060	1.74	1.42	0.06
aim-50-1-6-unsat-4	50	76	2	3	20	7,236,702	5,195,870	2,386,893	3,873,236	1.87	1.34	0.62
aim-50-2-0-sat-1	50	94	2	3	23	90,365,054	69,403,118	1,414,457	n/a	–	–	–
aim-50-2-0-sat-2	50	96	2	3	22	65,452,670	48,755,950	1,391,192	31,342,985	2.09	1.56	0.04
aim-50-2-0-sat-3	50	96	2	3	23	54,422,654	32,724,286	12,682,135	19,291,682	2.82	1.70	0.66
aim-50-2-0-sat-4	50	94	2	3	20	8,241,790	6,630,594	160,264	3,861,750	2.13	1.72	0.04
aim-50-2-0-unsat-1	50	97	2	3	26	606,571,518	433,412,038	6,546,400	n/a	–	–	–
aim-50-2-0-unsat-2	50	94	2	3	25	303,318,014	168,365,286	20,545,846	n/a	–	–	–
aim-50-2-0-unsat-3	50	92	2	3	24	170,571,646	116,379,982	7,669,688	n/a	–	–	–
aim-50-2-0-unsat-4	50	95	2	3	22	74,731,070	47,225,310	372,308	10,832,484	6.90	4.36	0.03
aim-50-3-4-sat-1	50	156	2	3	31	19,768,541,182	5,932,509,032	38,056,597	n/a	–	–	–
aim-50-3-4-sat-2	50	161	2	3	31	14,305,816,574	4,451,610,286	76,886	9,056,282	1580	492	0.01
aim-50-3-4-sat-3	50	161	2	3	29	7,761,709,054	3,734,762,630	1,906,662	n/a	–	–	–
aim-50-3-4-sat-4	50	159	2	3	31	19,749,318,654	5,467,090,270	348,265	13,478,054	1465	406	0.03
aim-100-1-6-sat-1	100	154	2	3	35	210,325,162,494	163,853,989,863	47,152,799,695	n/a	–	–	–
aim-100-1-6-sat-2	100	156	2	3	33	85,955,535,454	67,679,948,226	24,266,372,351	n/a	–	–	–
aim-100-1-6-sat-3	100	156	2	3	35	350,209,736,830	241,765,634,544	97,931,906,010	n/a	–	–	–

Table 2: Results for *AIM* benchmark instances. All problems have a median tightness ratio of $tr = 0.875$.

5 Impact of Orderings

In a separate experimental setup we wanted to investigate the power of our new bounds in predicting good orderings. We therefore looked at a handful of problem instances and evaluated the impact that the variable ordering for search can have on the bounds we compute, as well as on the true search space size.

To that end we processed the instances 50 times along a randomized minfill ordering (where ties are broken at random). Each time we computed the twb and hwb bound and solved the problem exactly, as described before.

In particular we examined four instances from the *AIM* class and four *SSA* circuit analysis instances. The results are plotted in Figure 4. Each vertical bar represents a single randomized run: the white top indicates the value of twb and the grey middle part denotes hwb, while the black bottom gives the true search space size $\#cm$ (if the grey part is not visible, hwb is very close to or the same as $\#cm$). In addition we record, against a different scale on the right hand side of the figures, the tree width of each ordering.

It is immediately obvious that both bounds provide a much better indicator of the quality of orderings than the plain tree width. For example, on the aim-50-1-6-sat-1 instance the tree width is 18 for most of the orderings, yet the values of $\#cm$ vary condiderably in these cases – but this variation is indeed captured by both the twb and hwb bound. Furthermore, some orderings have $w = 19$ and slightly increased twb value (columns 9 and 14, for instance), yet the search space size $\#cm$ is actually smaller than it is for many of the orderings with tree width 18 – a fact that is captured by a lower hwb bound.

Similar observations hold for other instances, for example aim-50-1-6-sat-3, where most orderings have the same tree width, but where twb and hwb are more informative

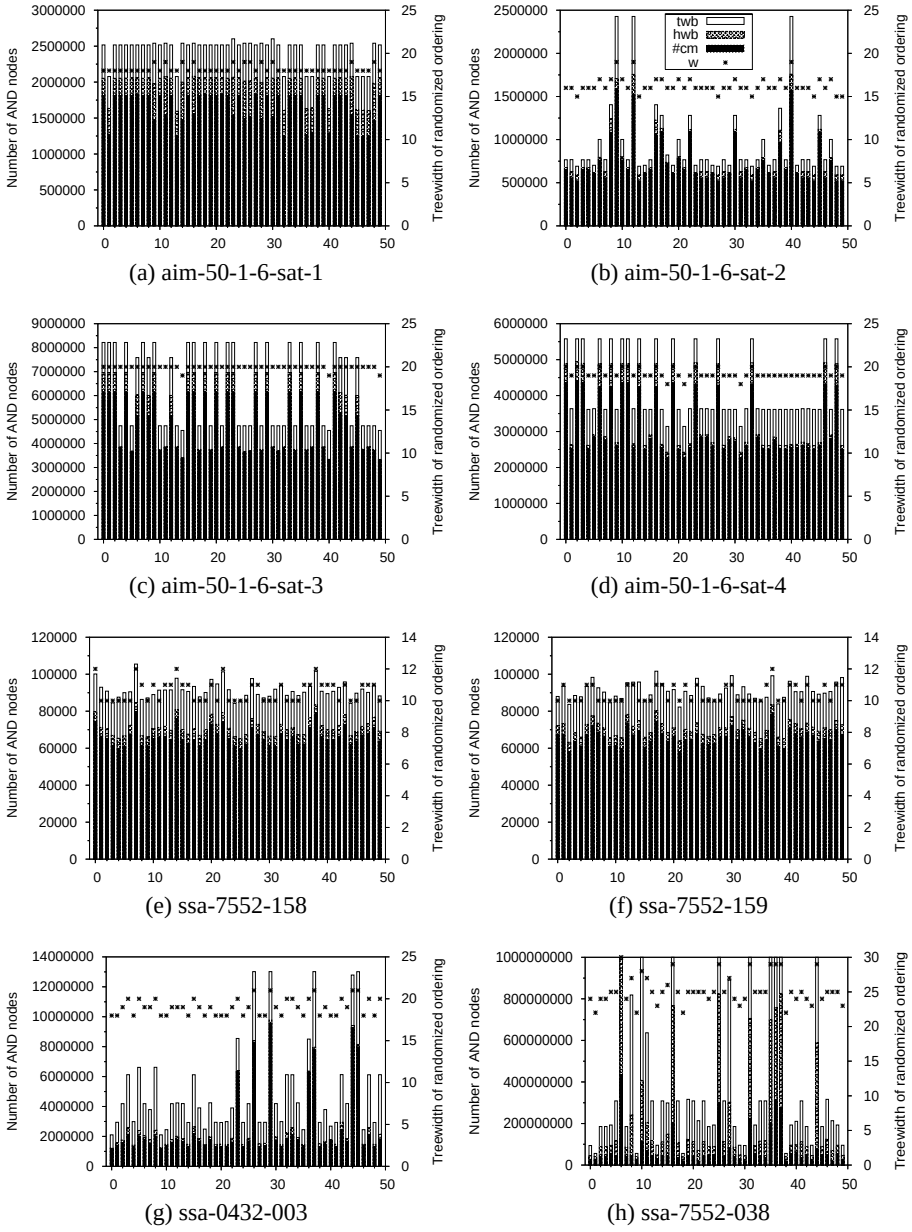


Fig. 4: Plots of the twb and hwb bounds versus the true search space size $\#cm$ on various problem instances, each over 50 randomized minfill variable orderings. Also shown is the tree width w for the each ordering, which is plotted against a separate scale on the right.

in determining "good" variable orderings. On aim-50-1-6-sat-4 twb is nearly the same for all orderings with tree width 19, yet hwb characterizes the distribution of different $\#cm$ values very well. For instance ssa-0432-003, iterations 38-41, neither the tree width nor the twb bound correctly identify the lowest $\#cm$, whereas hwb does.

Moreover, the diagrams once again exemplify the general improvement of the hwb bound over the twb one, as observed already in Section 4. It is also worth noting that hwb is a very tight bound in many of the cases in Figure 4.

6 Summary & Future Work

In this paper we describe schemes for upper and lower bounding the instance-based complexity of AND/OR search algorithms for solution counting and related $\#P$ problems. The typical asymptotic bound is exponential in the problem's tree width. While this can give a rough idea about problem hardness, it is often desirable to obtain a tighter, more fine-grained bound. As it has previously been shown, this can be accomplished by looking at a suitable tree decomposition of the problem's underlying graph structure and the domains of variables in the decomposition clusters. But this is blind to determinism in the problem, which can greatly prune the search space in practice.

The notion of determinism is, however, very prominent in the framework of hypertree decompositions which allows for exploiting tight relation representations; in practice, though, the bounds one obtains with this approach, exponential in hypertree width, are mostly not competitive and worse than those provided by the tree width.

To overcome this we initiate an effort in this paper and in an earlier one (under review) to develop more refined bounding methods that selectively exploit determinism in the relation specification. We demonstrated the potential of our first version of an upper bounding scheme on probabilistic networks for queries such as finding the probability of evidence. The contribution of the current paper is in:

1. Extending the upper bounding scheme to account not only for a determinism in each cluster independently, but also consider propagation of determinism that occurs when moving down the search tree, which has improved our bounds by 30% in some cases,
2. Developing a new approach for lower bounding the search space size,
3. A new set of empirical demonstrations of the power of both bounding schemes over a significant number of constraint network benchmarks, and
4. Demonstrating the ability of the bounds to discriminate between variable orderings for search, which can affect the performance substantially.

We believe that the current version of our upper bounding scheme can be further improved by incorporating a stronger form of propagation down the bucket tree, across clusters. Well-known consistency methods of varying complexity should be applicable for this. For optimization tasks and for approximating branch-and-bound and best-first search algorithms we hope to accomplish further tightening of the search space using the cost function itself. Finally, recent advances in sampling-based counting should allow us to improve the quality of the lower bounds we compute.

On a higher level, we also plan to extend our scheme to allow for flexible bound adaptation to reflect conditioning during search, which for example allows for dynamically updating the variable ordering; eventually we intend to deploy our scheme for parallelizing search algorithms over a network of many machines (e.g., grids and clusters), where the load balancing in partitioning the tasks can take advantage of a good estimate of each partition's workload.

References

1. R. Dechter and J. Pearl: Tree Clustering for Constraint Networks. *Artificial Intelligence* **38** (1989): 353–366.
2. R. Dechter and R. Mateescu: AND/OR search spaces for graphical models. In *Artificial Intelligence* **171** (2007): 73–106.
3. A. Dermaku, T. Ganzow, G. Gottlob, B. McMahan, N. Musliu, M. Samer: Heuristic Methods for Hypertree Decompositions. *Technical Report DBAI-TR-2005-53*, Vienna University of Technology, 2005
4. V. Gogate and R. Dechter: Approximate Counting by Sampling the Backtrack-free Search Space. In *Proceedings of AAAI'07*.
5. G. Gottlob, N. Leone, and F. Scarcello: A comparison of structural CSP decomposition methods. *Artificial Intelligence* **124** (2000): 243–282.
6. D. S. Johnson: Approximation algorithms for combinatorial problems. In *Proceedings of STOC'73*: 38–49.
7. K. Kask, R. Dechter, J. Larrosa, and A. Dechter: Unifying tree decompositions for reasoning in graphical models. *Artificial Intelligence* **166** (2005): 165–193.
8. U. Kjærulff: Triangulation of Graphs – Algorithms Giving Small Total State Space. *Research Report R-90-09, Dept. of Mathematics and Computer Science*, Aalborg University 1990.
9. S. L. Lauritzen and D. J. Spiegelhalter: Local Computations with Probabilities on Graphical Structures and Their Application to Expert Systems. *Journal of the Royal Statistical Society. Series B* **50(2)** (1988): 157–224.
10. A. K. Mackworth and E. C. Freuder: The Complexity of Some Polynomial Network Consistency Algorithms for Constraint Satisfaction Problems. *Artificial Intelligence* **25(1)** (1985): 65–74.
11. R. Mateescu and R. Dechter: The Relationship Between AND/OR Search Spaces and Variable Elimination. In *Proceedings of UAI'05*: 380–387.
12. L. Otten and R. Dechter: Bounding Search Space Size via (Hyper)tree Decompositions. Under review.
13. J. Pearl: Probabilistic Reasoning in Intelligent Systems. Morgan Kaufmann, 1988.
14. M. Silberstein, A. Tzemach, N. Dovgolevskiy, M. Fishelson, A. Schuster, D. Geiger: Online system for faster linkage analysis via parallel execution on thousands of personal computers. *American Journal of Human Genetics*, 2006.