Witnessing Solution Counting in Tree-Structured Methods for CSPs

Robert J. Woodward[1,2] Shant Karakashian[1]
Berthe Y. Choueiry[1] Christian Bessiere[2]

[1]Constraint Systems Laboratory
University of Nebraska-Lincoln, USA
{rwoodwar|shantk|choueiry}@cse.unl.edu

[2]LIRMM-CNRS
University of Montpellier, France
bessiere@lirmm.fr

TR-UNL-CSE-2016-0006

October 27, 2016

Abstract

Counting the exact number of solutions of a Constraint Satisfaction Problem (CSP) is an important but difficult task. To overcome this difficulty, the techniques proposed in the literature organize the search process along a tree decomposition of the CSP, where all the extensions of a given partial solution over different branches of the tree are first independently counted in each branch before their numbers can be multiplied. We observe that this count is zero when any of the branches has no solution. We propose witness-based search, which first ensures the existence of a solution (i.e., witness) in each branch before starting the counting. We empirically establish the benefits of our technique in the context of the BTD and AND/OR search graphs.

Contents

1 Introduction

Counting the number of solutions of a Constraint Satisfaction Problem (CSP), an important task in verification and automated reasoning, is known to be #P-complete [Valiant, 1979]. Current techniques for solving this problem exploit some *tree structure* of the constraint network of the CSP in order to reduce the search and counting efforts [Dechter and Pearl, 1988; Gogate and Dechter, 2008; Favier *et al.*, 2009].

Indeed, in a tree-structured problem, the number of solutions at any node in the tree is computed by simple algebraic operations (i.e., summation and product) from the number of solutions of the children of the node and information at the node itself, following a pre-order traversal. In a non-parallel implementation, all the solutions in one branch of the tree are counted before the solutions in another branch with the same parent. In case the latter branch has no solution, the effort spent counting the solutions in the first branch are wasted. We propose to first *find* a witness solution in every branch of a given node in the tree before proceeding to *counting* the number of solutions in any given branch. We call this scheme *witness-based search*.

Further, tree-structured methods typically and heavily exploit a caching mechanism. This mechanism maintains, at some nodes of the search space, results that were derived during search in order to reduce the amount of repetitive and redundant work done during search. The information cached includes (portions of) partial solutions that yielded inconsistencies (i.e., nogoods) and also those that yielded solutions (i.e., goods) along with the count of solutions found.

We apply witness-based search to two solution-counting methods, namely, the Backtrack Search with Tree Decomposition (BTD) [Jégou and Terrioux, 2003] and the AND/OR search tree [Dechter and Mateescu, 2004]. Our empirical evaluations show a reduction of the search effort, and, importantly, the space used for caching, which is a major bottleneck in those techniques.

This paper is structured as follows. Section 2 recalls main concepts and definitions. Section 3 discusses solution-counting methods based on tree structures. Section 4 describes and discusses witness-based solution counting. Section 5 describes our experiments, and Section 6 concludes.

2 Main Definitions

We first summarize the main concepts and definitions used.

2.1 Constraint Satisfaction Problem

A *Constraint Satisfaction Problem* (CSP) is defined by $(\mathcal{X}, \mathcal{D}, \mathcal{C})$, where $\mathcal{X}$ is a set of variables, $\mathcal{D}$ is a set of domains, and $\mathcal{C}$ is a set of constraints. Each variable in $\mathcal{X}$ has a finite domain in $\mathcal{D}$, and is constrained by a subset of the constraints in $\mathcal{C}$. Each constraint $C_i \in \mathcal{C}$ is defined by a relation R_i specified over the *scope* of the constraint, *scope*(C_i), which are the variables to which the constraint applies, as a subset of the Cartesian product of the domains of those variables. A tuple $t_i \in R_i$ is a combination of values for the variables in the scope of the constraint that is either allowed (i.e., support) or forbidden (i.e., conflict). A solution to the CSP is an assignment to each variable of a value taken from its domain such that all the constraints are satisfied. In general, finding a solution to a CSP is NP-complete, and counting its number of solutions is #P-complete.

Backtrack search is a sound and complete algorithm commonly used to solve CSPs. To improve the performance of search and reduce the severity of the combinatorial explosion, we enforce a given local consistency level. One common such property is Generalized Arc Consistency (GAC). A CSP is GAC iff for every constraint, any value in the domain of any variable in the scope of the constraint can be extended to a tuple satisfying the constraint.

Several graphical representations of a CSP exist. In the *hypergraph*, the vertices represent the variables of the CSP, and the hyperedges represent the scopes of the constraints (see Figure 1). In the *primal graph*, the vertices represent the CSP variables, and the edges connect every two variables that appear in the scope of some constraint (see Figure 2).

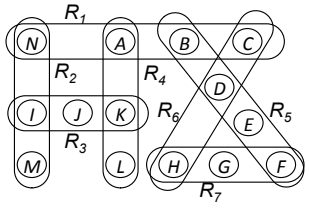

Figure 1: A hypergraph

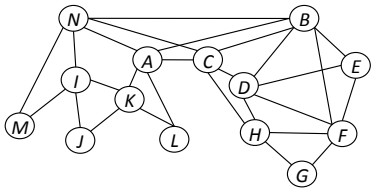

Figure 2: The primal graph

2.2 Backtrack Search with Tree Decomposition

A *tree decomposition* of a CSP is a tree embedding of its constraint network. The tree nodes are *clusters* of variables and constraints from the CSP. The set of

variables of a cluster cl is denoted $\chi(cl) \subseteq \mathcal{X}$, and the set of constraints $\psi(cl) \subseteq \mathcal{C}$. A tree decomposition must satisfy two conditions:

1. Each constraint appears in at least one cluster and the variables in its scope must appear in this cluster; and

2. For every variable, the clusters where the variable appears induce a connected subtree.

Many techniques for generating a tree decomposition of a CSP exist [Dechter and Pearl, 1989; Jeavons *et al.*, 1994; Gottlob *et al.*, 2000]. We use here the tree-clustering technique [Dechter and Pearl, 1989]. *First*, we triangulate the primal graph of the CSP using the min-fill heuristic [Kjærulff, 1990]. *Second*, using the perfect elimination ordering given by the MAXCARDINALITY algorithm [Tarjan and Yannakakis, 1984], we identify the maximal cliques in the resulting chordal graph using the MAXCLIQUES algorithm [Golumbic, 1980], and use the identified maximal cliques to form the clusters of the tree decomposition. Figure 3 shows a triangulated primal graph of the example in Figure 1. The dotted edges (B,H) and

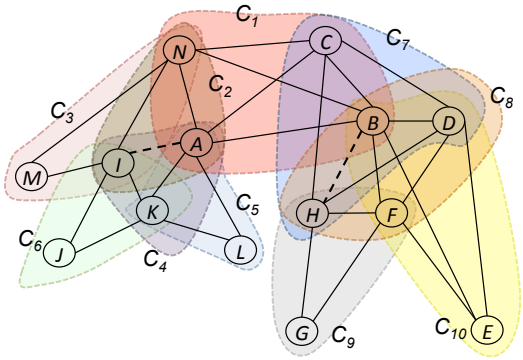

Figure 3: Triangulated primal graph and its maximal cliques

(A,I) in Figure 3 are fill-in edges generated by the triangulation algorithm. The ten maximal cliques of the triangulated graph are highlighted with 'blobs.' *Third*, we build the tree by connecting the clusters using the JOINTREE algorithm [Dechter, 2003]. While any cluster can be chosen as the root of the tree, we choose the cluster that minimizes the longest chain from the root to a leaf. Figure 4 shows the tree after connecting the maximal cliques of Figure 3. *Finally*, we determine the variables and constraints of each cluster as follows: *a*) The variables of a cluster cl, $\chi(cl)$, are the variables in the maximal clique that yields the cluster;

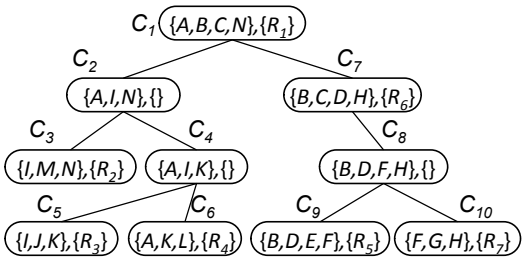

Figure 4: A tree decomposition of the CSP in Figure 1

and *b*) The constraints of a cluster cl, $\psi(cl)$, are all the constraints R_i, such that *scope*$(R_i) \subseteq \chi(cl)$. Figure 4 shows a tree decomposition for the example of Figure 1. Note that we may end up with clusters with no constraints (e.g., C_2, C_4 and C_8). A *separator* of two adjacent clusters is the set of variables that are associated with both clusters.

2.3 AND/OR Tree Search

AND/OR tree search was proposed by Dechter [2004] as a generalization of search in graphical models. AND/OR tree search exploits (in)dependencies in the model to exponentially reduce the search effort, binding it exponentially by, instead of the number of variables, the depth of a *pseudo-tree* [Freuder and Quinn, 1987], which is a tree spanning of the model. Dechter also extended the AND/OR search space from a tree to a *graph*, further reducing the time effort albeit at the cost of increased memory space [2004]. The detailed definitions and characterizations are accessible in the original papers; below we illustrate this process with a simple example.

Consider a CSP with the constraint graph shown in Figure 5. The domain of

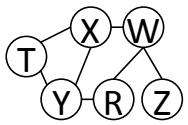

Figure 5: A constraint graph

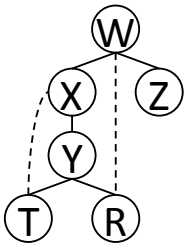

Figure 6: A pseudo-tree of the example from Figure 5

the variable Y is $\{2, 3\}$. The domains of W, X, Z, T, R are $\{1, 2\}$. The constraints are as follows: $W = Z$, $W = R$, $W \geq X$, $0 \leq T - X \leq 1$, and $X < Y$. The constraint between Y and T forbids only the tuple $\langle (Y, 2), (T, 1) \rangle$. Similarly, the constraint between Y and R forbids only the tuple $\langle (Y, 2), (R, 1) \rangle$). Figure 6 gives a pseudo-tree of this CSP where the dependencies between variables are shown as the tree edges (full lines) and back-edges (dotted lines). Figure 7 shows the AND/OR search tree of the example in Figure 5 using the pseudo-tree of Figure 6. An AND/OR search tree alternates between OR nodes (variables) and

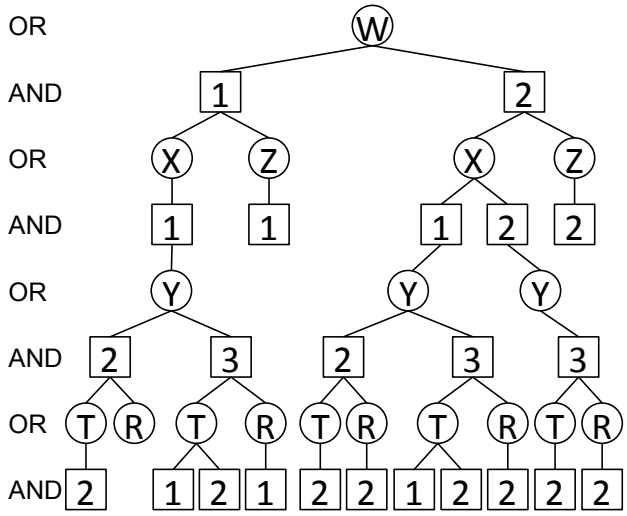

Figure 7: An AND/OR search tree of the example from Figure 5

AND nodes (variable assignments). The structure of the AND/OR search tree is based on the pseudo-tree. The root of the AND/OR search tree is an OR node for the variable at the root of the pseudo-tree. The children of an OR node are AND nodes corresponding to the value assignments of the variable of the OR node. The children of an AND node are OR nodes, corresponding to the variables that are the children of the AND node's variable in the pseudo-tree.

The *parents* of an OR node V are the ancestors of V in the pseudo-tree that are connected in the constraint graph to V or to descendants of V. The *parent-separator* of an OR node V (or an AND node $\langle V, v \rangle$) is the set containing V and its ancestors in the pseudo-tree that are connected in the original graph to descendants of V. The *context of an* AND *node* is the assignments of the variables in the node's parent-separator. The *context of an* OR *node* is the assignments of the variables in the node's parents. Two nodes can be merged together if their context is the

same, thus yielding a search graph. Figure 8 shows the AND/OR search graph of our example using OR context-merging. Note that we could merge nodes on *both*

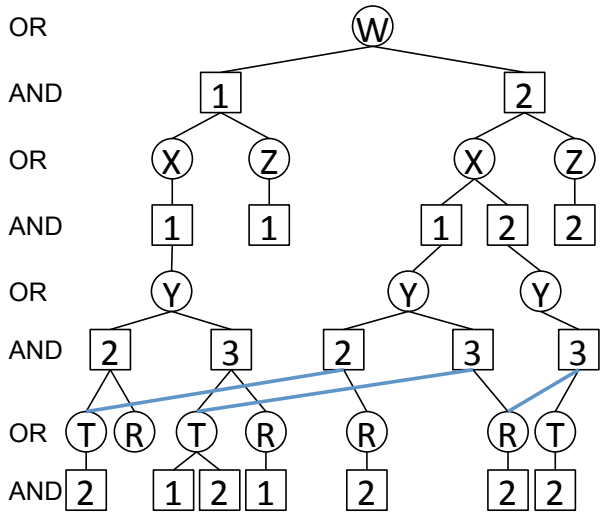

Figure 8: The AND/OR search graph by merging OR contexts

the OR context and AND context; however, merging with one context makes the other unnecessary [Dechter and Mateescu, 2006].

3 Tree-Based Solution Counting

Below, we discuss solution-counting methods and provide a pseudo-code that operates on binary tree-structured CSPs, the BTD, and AND/OR search graphs. In Section 4, we modify this pseudo-code to incorporate our witness mechanism. Our pseudo-code is specified recursively for readability, but our implementation is iterative. Further, the pseudo-code relies on back-checking for extending consistent partial solutions, whereas our implementation uses look-ahead.

3.1 Solution Counting in a Tree-Structured Binary CSP

Dechter and Pearl [1988] noted that the number of solutions in a tree-structured binary CSP can be computed in $O(nd^2)$ where n is the number of variables and d the maximum domain size. It computes the number of solutions of a given CSP variable from the number of solutions of its children in the tree. In summary,

1. the number of solutions rooted at a given variable in the tree-structured CSP is the summation of the number of solutions 'rooted' at each value in the domain of the variable; and,

2. the number of solutions at a given value of the domain is the product of the numbers of solutions of the value's consistent extensions in each of the children of the variable.

We wrote Algorithm 1 to loosely accommodate all three solution methods discussed in this section. The algorithm is started by running #SOLS(*root*,$\emptyset$), where *root* is the root of the tree. $SolCache(child, \mathcal{A})$ is the cache of a node given a partial assignment $\mathcal{A}$, and stores, when *bound*, the number of solutions rooted at the node. n_{total} stores the number of solutions at the $root$, and n_c stores the number of solutions rooted at the assignment $root \leftarrow v$. Whenever $n_c = 0$ within the loop of Lines 4–11, we exit the loop. This test is omitted for readability. The original procedure of Dechter and Pearl [1988] is easily obtained by ignoring the cache (Lines 5, 6, 7, 9, and 10).

Algorithm 1: #SOLS($root$,$\mathcal{A}$)

Input: $root$ of a tree structure of a CSP
$\mathcal{A}$: A current partial solution
Output: Number of solutions at $root$

```
1  n_total ← 0
2  foreach v ∈ Domain(root) s.t. v is consistent with A do
3      n_c ← 1; A_cur ← A ∪ {root ← v}
4      foreach child ∈ Children(root) do
5          if SolCache(child, A_cur) is bound then
6              cache ← SolCache(child, A_cur)
7          else
8              cache ← #SOLS(child, A_cur)
9              SolCache(child, A_cur) ← cache
10             Cache good, no-good
11         n_c ← n_c × cache
12     n_total ← n_total + n_c
13 return n_total
```

3.2 Solution Counting in the BTD

In the case of the BTD, Algorithm 1 operates on a tree decomposition of the CSP. Line 2 is called on the last unassigned variable in the root cluster as $root$. The child in Line 4 is the first unassigned variable in a child cluster. Before the recursive call in Line 8 is done on the last unassigned variable in the child cluster, we must first consistently extend the partial solution over the unassigned variables in the child cluster except for one variable.

When search succeeds, the BTD caches the instantiation of the variables at the separators as a 'good' along with the number of solutions rooted at this instantiation. Otherwise, the instantiations at the separator is cached as a 'no-good.'

3.3 Solution Counting in an AND/OR Search Tree

In the case of an AND/OR search tree, Algorithm 1 operates on the pseudo-tree. To count the solutions, the AND nodes multiply the numbers of solutions of their children (leaf AND nodes are considered to have one solution); OR nodes add the number of solutions of their children (leaf OR nodes are considered to have 0 solutions).

The cached information is similar to that cached by the BTD, except that it is for the instantiations of the variables in the contexts (not at the separators). The performance of this method is improved by the detection of dead-caches, which are caches that will never be hit [Darwiche, 2001; Marinescu and Dechter, 2006], and, thus, need not be recorded. In the presence of a dead-cache, the assignment of $SolCache(child, \mathcal{A})$ in Line 9 is not executed, and goods/no-goods are not stored in Line 10. Note that, the space needed for caching is a major bottleneck in tree-based solution-counting methods. Other techniques for dealing with this bottleneck exist (e.g., naive-caching and adaptive-caching [Marinescu and Dechter, 2006]) and are orthogonal to our approach.

4 Solution Counting in Witness-Based Search

The idea of our witness-based search is to refrain from counting solutions in any branch off a node in a tree structure before ensuring that the current partial solution at the node[1] can be consistently extended over the variables in *each* branch

[1] An AND node in the case of an AND/OR search tree.

off the node. Indeed, if the solution fails to extend consistently over the variables of a single branch, then all the counting effort in the branches is wasted by multiplication by 0.

4.1 A Generic Pseudo-Code for Witness-Based Search

We specify witness-based search in the generic pseudo-code of Algorithm 2, and claim that it is applicable to any tree-based solution-counting method. Our technique interacts too tightly with the solution-counting strategies for it to be implemented as a separate component of a Constraint Solver. Indeed, the caching information stored by the various solution-counting strategies depend on the strategy itself. Based on our experience with two such strategies (i.e., BTD and AND/OR tree search), we found that the code of such strategies must be directly modified to *incorporate* witness-based search.

Algorithm 2 differs from Algorithm 1 by the use of a switch variable $mode$, which takes one of two values *sat* or *count* to determine whether search should check for satisfiability (i.e., find a witness) or do solution counting, respectively. The algorithm is started by running W#SOLS($root$,$\emptyset$,*count*), where $root$ is the root of the tree. In Line 2, the algorithm examines all the children, either finding a witness in the cache (Line 5) or doing the search to find a witness (Line 8). If $mode$=*sat*, then 1 is returned (Line 12). If $mode$=*count* and a witness is found, the algorithm proceeds to counting the number of solutions (Lines 14 to 23). Comparing Line 10 and Line 21 only goods are cached when $mode$=*count* because satisfiability is guaranteed by the witness mechanism.

4.2 Analysis of Witness-Based Search

In order to save on the search effort, the implementation of the algorithm should preserve the state of the search space in a branch where a witness is found so that, when the same branch is revisited again to count the remaining solutions, the effort to find the first solution is not repeated and the search can proceed from the witness. Below, we discuss two implementation strategies for handling the state of the search space where a witness solution was found. The first strategy does not always preserve the state of this space, whereas the second does.

In the first implementation strategy, after finding a witness in a branch br_i, we maintain the instantiations of the variables in this branch (i.e., freeze the search space in br_i) while checking on the other branches (which are independent of br_i). Thus, the recursive call in Line 19 to count solutions in br_i can continue

Algorithm 2: W#SOLS($root$,$\mathcal{A}$,$mode$)

Input: $root$ of a tree structure of a CSP
$\mathcal{A}$: A current partial solution
$mode$: Either *sat* for satisfiability or *count* for solution counting

Output: If *mode*=*count*, number of solutions at $root$. Otherwise (*mode*=*sat*), 1 if a witness was found, 0 otherwise

```
1  n_total ← 0
2  foreach v ∈ Domain(root) s.t. v is consistent with A do
3      n_c ← 1; A_cur ← A ∪ {root ← v}
4      foreach child ∈ Children(root) do
5          if SolWitnessCache(child, A_cur) is bound then
6              cache ← SolWitnessCache(child, A_cur)
7          else
8              cache ← W#SOLS(child, A_cur, sat)
9              SolWitnessCache(child, A_cur) ← cache
10             Cache good, no-good
11         n_c ← n_c × cache
12     if mode=sat and n_c > 0 then return 1
13     if mode=count and n_c > 0 then
14         n_c ← 1
15         foreach child ∈ Children(root) do
16             if SolCache(child, A_cur) is bound then
17                 cache ← SolCache(child, A_cur)
18             else
19                 cache ← W#SOLS(child, A_cur, count)
20                 SolCache(child, A_cur) ← cache
21                 Cache good
22             n_c ← n_c × cache
23         n_total ← n_total + n_c
24 return n_total
```

from the current (frozen) state of the search. However, when backtracking occurs in the search above br_i, the variables in br_i and up to the backtrack level are uninstantiated (i.e., the search space in br_i is reset). When search resumes, and

if the current path 'conditions' br_i in the same way as it did earlier,[2] we know, because of the stored good, that br_i has a witness. However, the state of the search space in br_i was reset because of backtracking. Thus, solution counting will have to restart from scratch. The advantage of this implementation is that it does not add to the memory space requirements. Its disadvantage is that, upon backtracking, the effort to find this first-solution has to be repeated.

The second implementation strategy is similar to the first, except that the caching is enhanced to also store the variable-value assignments of the witness (i.e., the first solution in br_i). Thus, upon backtracking, the state of the search space in br_i is restored and search can continue from that state when counting the number of solutions in br_i (Line 19). The advantage of this strategy is that the effort to find the first-solution need not be repeated. However, the storage size for each cached good is increased linearly in the number of the variables in the branch.

While the first implementation strategy cannot guarantee that witness-based search does not increase the number of nodes visited by search, the second strategy does. We implemented both strategies: the first for the BTD, and the second for AND/OR tree search. We found them both to be advantageous on the tested instances despite the occasional and slight increase in the number of nodes visited by the witness-based BTD.

5 Empirical Evaluations

Our experiments assess the improvement brought about by the witness mechanism on solution-counting methods. We show that adding witness to both the BTD and AND/OR tree search results in significant improvements of both time and space on both unsatisfiable and satisfiable CSP instances.

5.1 Experimental Set-Up

We integrate GAC (GAC2001 [Bessière *et al.*, 2005]) in all our search algorithms as a real full look-ahead strategy. We find the pseudo-tree using the technique described by Bayardo and Mirankar [1996]. We instantiate the variables in the order of the pseudo-tree.

[2]Determined by the instantiations of the variables in the separators/contexts.

The experiments are conducted on the benchmarks of the CSP Solver Competition[3] with a time limit of two hours per instance and 8 GB of memory. We provide plenty of time and memory, to the extent possible, to avoid tainting our experiments with censored data. We use benchmarks[4] that are difficult for BTD and AND/OR tree search to illustrate the advantage of using the witness technique in a challenging context. We split our analysis on the 479 unsatisfiable and 200 satisfiable instances tested.

It is not our goal to compare the performances of BTD with AND/OR tree search, but to evaluate the improvement brought about by witness-based search on each of them. For each of the two solution-counting methods, we focus our analysis on instances that were completed by search with and without the witness technique. Of the original 679 instances, the number of those instances is 308 for BTD and 239 for AND/OR search tree.[5] Further, we ignore the instances where the performance did not change in terms of nodes visited (on those instances the CPU time difference was within *less than 0.1%* and they used the same caching space). We end up with 106 instances for BTD and 95 instances for AND/OR search tree.[6] We analyze the performance by reporting the following measurements: *a*) the number of nodes visited, *b*) the CPU run time in seconds, and *c*) the space requirement in terms of number of goods and no-goods stored. The information about the witness needed to restore the state of the search space is included in the goods measurement. We show that witness-based search is advantageous by all three measurements.

5.2 Comparing Witness-BTD with BTD

In Tables 1-3, we abbreviate Witness-BTD as W-BTD.

Number of nodes visited: Table 1 shows the number of instances that a given technique visits fewer nodes than the other, and the average number of nodes vis-

[3] `http://www.cril.univ-artois.fr/CPAI08/`

[4] aim-(50, 100, 200), composed-(25-10-20, 25-1-2, 25-1-25, 25-1-40, 25-1-80, 75-1-2, 75-1-25, 75-1-40, 75-1-80), dag-rand, dubois, graphColoring-(hos, mug, register-mulsol, register-zeroin, sgb-book, sgb-games, sgb-miles, sgb-queen), hanoi, modifiedRenault, QCP-15, rand-(10-20-10, 8-20-5), rlfap(GraphsMod, Scens11, ScensMod), ssa, and tightness0.9

[5] For BTD: 197 unsatisfiable, 111 satisfiable. For AND/OR search tree: 155 unsatisfiable, 84 satisfiable.

[6] For BTD: 69 unsatisfiable, 37 satisfiable. For AND/OR search tree: 59 unsatisfiable, 36 satisfiable.

ited by each algorithm. Note that BTD *never* outperforms Witness-BTD on un-

Table 1: Number of instances with fewer #NV, and average #NV

	BTD	W-BTD
Fewest #NV		
UNSAT (69)	0	**69**
SAT (37)	8	**29**
Average #NV		
UNSAT (69)	1,431,275.77	**616,502.46**
SAT (37)	8,235,685.41	**8,166,271.57**

satisfiable instances. On satisfiable instances, Witness-BTD wins more often than BTD (29 instances). However, there are instances where BTD visits fewer nodes than Witness-BTD (8 instances). The reason is because the implementation of Witness-BTD does not restore the search space for cached witnesses, but instead searches again for the first solution, as discussed in Section 4.2. Witness-BTD clearly outperforms BTD on unsatisfiable instances, showing substantial savings of not searching partial solutions that never participate in a global solution. On satisfiable instances, the difference is not as significant, albeit it shows an improvement. Notice, that although some search effort was wasted in our implementation of Witness-BTD (BTD visited fewer nodes on 8 instances than Witness-BTD), Witness-BTD still always saves on average on the number of nodes visited.

Run time: The savings in the number of nodes visited match those exhibited by the CPU time. Table 2 reports the number of instances on which a given algorithm completed fastest within the CPU clock-resolution of 100 ms (thus, with occasional ties), and the average CPU time. On unsatisfiable instances, Witness-BTD solves more instances fastest than BTD and has a smaller average CPU time. On satisfiable instances, BTD is fastest on more instances than Witness-BTD (21 vs. 17 instances). However, the average CPU time is slightly less for the Witness-BTD. Thus, Witness-BTD yields savings (on unsatisfiable instances) while causing no significant overhead (on satisfiable instances).

Space requirements: Table 3 gives the average number of stored goods and no-goods. Notice that the number of no-goods for Witness-BTD and BTD are almost

Table 2: #Instances completed fastest and average time

	#Fastest		Avg. time (sec.)	
	BTD	W-BTD	BTD	W-BTD
UNSAT (69)	21	**55**	135.28	**110.01**
SAT (37)	**21**	17	724.51	**723.97**

Table 3: Average number of goods and no-goods stored

	BTD	W-BTD
Average #no-goods		
UNSAT(69)	43,675.77	**43,675.74**
SAT(37)	449,633.21	**449,516.29**
Average #goods		
UNSAT(69)	24,104.52	**10,611.10**
SAT(37)	160,025.63	**148,739.58**

identical, which is to be expected given that Witness-BTD finds the same no-goods, only earlier. *However, the number of goods stored is significantly reduced by* Witness-BTD. This fact illustrates how Witness-BTD avoids storing partial solutions that cannot be completed to global solutions, which is exactly our intended design.

In summary, Witness-BTD achieves its goal: it saves on the number of nodes visited, time, and space, and never yields any overheads. It is a safe and robust strategy to implement in all circumstances, and clearly improves BTD. Therefore, it can be safely applied at all times.

5.3 Comparing Witness-AND/OR with AND/OR Tree Search

In Tables 4-6, we abbreviate AND/OR tree search as AO and witness-AND/OR tree-search as W-A/O.

Number of nodes visited: As stated in Section 4.2, Witness-AND/OR tree search is guaranteed to never visit more nodes than AND/OR tree search does.

The variable-value assignments of the witness are cached so that the state of the search space can be restored to allow solution counting to resume from the witness. Table 4 gives the average number of nodes visited by each strategy and shows a large reduction on both satisfiable and unsatisfiable instances.

Table 4: Average #NV

	A/O	W-A/O
UNSAT (59)	580,762.02	**537,552.56**
SAT (36)	24,314,616.44	**19,521,667.08**

Run time: Once again, the reduction of the nodes visited directly translates into CPU time savings. Table 5 shows the number of instances on which a given algorithm completed the fastest (within the CPU clock-resolution) and the average CPU time. AND/OR tree search did complete a few instances fastest (19 unsatisfiable and 10 satisfiable). However, note that the 19 unsatisfiable instances that AND/OR tree search completed fastest *tie* with Witness-AND/OR tree search. Indeed, Witness-AND/OR tree search was fastest on all 59 unsatisfiable instances. Looking at the average CPU time, Witness-AND/OR outperformed AND/OR tree search on both satisfiable and unsatisfiable instances.

Table 5: #Instances completed fastest and average time

	#Fastest		Avg. time (sec.)	
	A/O	W-A/O	A/O	W-A/O
UNSAT (59)	19	**59**	110.56	**102.96**
SAT (36)	10	**29**	693.16	**569.73**

Space requirements: Table 6 gives the average number of stored goods and no-goods by AND/OR and Witness-AND/OR tree search. As discussed for the case of BTD, there are roughly the same number of no-goods stored for Witness-AND/OR and AND/OR tree search. However, the average number of goods stored

Table 6: Average number of goods and no-goods stored

	A/O	W-A/O
Average #no-goods		
UNSAT(59)	9,645.76	**9,645.66**
SAT(36)	725,561.92	**711,190.75**
Average #goods		
UNSAT(59)	8,103.34	**5,783.95**
SAT(36)	103,506.00	**47,470.53**

is significantly reduced on both satisfiable and unsatisfiable instances. Because witness-based search dramatically reduces the space needed for caching, it directly benefits adaptive caching schemes to maintain more information cached than it would otherwise be possible [Marinescu and Dechter, 2006].

In summary, Witness-AND/OR tree search is a beneficial strategy to implement and use in all circumstances and clearly improves AND/OR tree search.

5.4 An example with extreme benefits

While the average values of the results reported above show a clear advantage of the witness-based search, we explore below a situation where an extreme saving can be obtained.

Inspired by the experiments reported by Otten and Dechter [2012], we manually create an instance of a CSP that has a very large search space but is unsolvable. We show how Witness-AND/OR search can yield extreme gains. In practice, we proceed as follows. We connect a large search space many solutions to another search space with no solutions as illustrated in Figure 9. To this end, we generate the pseudo-tree of each problem independently. We identify the root node of the barren search space. In the pseudo-tree of the solvable instance, we identify a variable that appears at the 'middle height' of the tree. Then, we add an arbitrary binary constraint between the two identified variables, thus linking the two CSP instances. We solve the newly formed instance with both AND/OR and Witness-AND/OR.

We generated one such problem by connecting an unsatisfiable aim-50 instance (normalized-aim50-1-6-unsat1.xml) to a pseudo-garden instance (normalized-

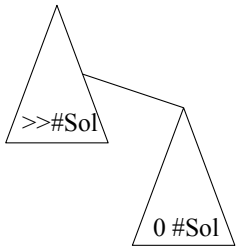

Figure 9: Connecting a tree with no solution to a tree with many solutions

g-9x9.xml) by adding an equality constraint between two variables (V53 of pseudo-garden to V1 of aim-50). The results were as follows:

1. AND/OR search expanded 2,657,758 nodes and detected unsolvability in 31.61 seconds.

2. Witness-AND/OR search reduces the effort by over 90%, visiting 63,476 nodes for a total of 2.25 seconds CPU time.

This example illustrates the significant advantage witness-based techniques can provide. Again, as stated earlier, this advantage does not cause any overhead.

6 Conclusions

In this paper, we proposed witness-based search as a strategy to improve the time and space performance of solution-counting methods that operate on a tree structure. We empirically showed that our technique benefit solution-counting methods based on the BTD and AND/OR tree search improving performance by all measurements, especially the space needed for caching, which is a major bottleneck in such methods. As future work, we plan to extend our approach to approximate solution counting [Gogate and Dechter, 2008]. We believe that the space savings obtained by our witness strategy will allow us to achieve better approximations.

Acknowledgments:

The authors gratefully acknowledge the guidance and feedback of Rina Dechter. This research is supported by NSF Grant No. RI-111795 and RI-1619344. Experiments were conducted on the equipment of the Holland Computing Center at the University of Nebraska-Lincoln.

References

[Bayardo and Mirankar, 1996] Roberto J. Bayardo and Daniel P. Mirankar. A Complexity Analysis of Space-Bound Learning Algorithms for the Constraint Satisfaction Problem. In *Proceedings of the Thirteen National Conference on Artificial Intelligence (AAAI 1996)*, pages 298–304, 1996.

[Bessière *et al.*, 2005] Christian Bessière, Jean-Charles Régin, Roland H.C. Yap, and Yuanlin Zhang. An Optimal Coarse-Grained Arc Consistency Algorithm. *Artificial Intelligence*, 165(2):165–185, 2005.

[Darwiche, 2001] Adnan Darwiche. Recursive Conditioning. *Artificial Intelligence*, 126(1-2):5–41, 2001.

[Dechter and Mateescu, 2004] Rina Dechter and Robert Mateescu. The Impact of AND/OR Search Spaces on Constraint Satisfaction and Counting. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP 2004)*, volume 3258 of *Lecture Notes in Computer Science*, pages 731–736. Springer, 2004.

[Dechter and Mateescu, 2006] Rina Dechter and Robert Mateescu. AND/OR Search Spaces for Graphical Models. *Artificial Intelligence*, 171(2-3):73–106, 2006.

[Dechter and Pearl, 1988] Rina Dechter and Judea Pearl. Network-Based Heuristics for Constraint-Satisfaction Problems. *Artificial Intelligence*, 34:1–38, 1988.

[Dechter and Pearl, 1989] Rina Dechter and Judea Pearl. Tree Clustering for Constraint Networks. *Artificial Intelligence*, 38:353–366, 1989.

[Dechter, 2003] Rina Dechter. *Constraint Processing*. Morgan Kaufmann, 2003.

[Dechter, 2004] Rina Dechter. AND/OR Search Spaces for Graphical Models. Technical report, University of California, Irvine, 2004.

[Favier *et al.*, 2009] Aurélie Favier, Simon de Givry, and Philippe Jégou. Exploiting Problem Structure for Solution Counting. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP 09)*, volume 5732 of *Lecture Notes in Computer Science*, pages 335–343, 2009.

[Freuder and Quinn, 1987] Eugene C. Freuder and Michael J. Quinn. The Use of Lineal Spanning Trees to Represent Constraint Satisfaction Problems. Technical Report 87-41, University of New Hampshire, 1987.

[Gogate and Dechter, 2008] Vibhav Gogate and Rina Dechter. Approximate Solution Sampling (and Counting) on AND/OR Spaces. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP 2008)*, volume 5202 of *Lecture Notes in Computer Science*, pages 534–538. Springer, 2008.

[Golumbic, 1980] Martin C. Golumbic. *Algorithmic Graph Theory and Perfect Graphs*. Academic Press Inc., New York, NY, 1980.

[Gottlob *et al.*, 2000] Georg Gottlob, Nicola Leone, and Francesco Scarcello. A Comparison of Structural CSP Decomposition Methods. *Artificial Intelligence*, 124(2):243–282, 2000.

[Jeavons *et al.*, 1994] Peter G. Jeavons, David A. Cohen, and Marc Gyssens. A Structural Decomposition for Hypergraphs. *Contemporary Mathematics*, 178:161–177, 1994.

[Jégou and Terrioux, 2003] Philippe Jégou and Cyril Terrioux. Hybrid Backtracking Bounded by Tree-Decomposition of Constraint Networks. *Artificial Intelligence*, 146:43–75, 2003.

[Kjærulff, 1990] U. Kjærulff. Triagulation of Graphs - Algorithms Giving Small Total State Space. Research Report R-90-09, Aalborg University, Denmark, 1990.

[Marinescu and Dechter, 2006] Radu Marinescu and Rina Dechter. Memory Intensive Branch-and-Bound Search for Graphical Models. In *Proceedings of the Twenty-First National Conference on Artificial Intelligence (AAAI 2006)*, pages 1200–1205, 2006.

[Otten and Dechter, 2012] Lars Otten and Rina Dechter. Anytime AND/OR Depth-first Search for Combinatorial Optimization. *AI Communications*, 25(3):211–227, 2012.

[Tarjan and Yannakakis, 1984] Robert Endre Tarjan and Mihalis Yannakakis. Simple Linear-Time Algorithms to Test Chordality of Graphs, Test Acyclicity

of Hypergraphs, and Selectively Reduce Acyclic Hypergraphs. *SIAM Journal on Computing*, 13(3):566–579, 1984.

[Valiant, 1979] Leslie G. Valiant. The Complexity of Computing the Permanent. *Theoretical Computer Science*, 8:189–201, 1979.