Improving Exact Solution Counting for Decomposition Methods

Philippe Jégou Hanan Kanso Cyril Terrioux
Aix-Marseille Univ, Université de Toulon, CNRS, ENSAM, LSIS, Marseille, France
{philippe.jegou, hanan.kanso, cyril.terrioux}@lsis.org

***Abstract*—The problem of counting solutions in CSP, called #CSP, is an extremely difficult problem that has many applications in Artificial Intelligence. This problem can be addressed by exact methods, but more classically it is solved by approximate methods. Here, we focus primarily on the exact counting. We show how it is possible to improve the methods based on structural decomposition by offering to enhance the search for a new solution which is a critical step for counting, particularly for such methods. Moreover, if the resources in time or in space are insufficient, we show that our approach is still able to provide a lower bound of the result. Experiments on CSP benchmarks show the practical advantage of our approach w.r.t. the best methods of the literature.**

***Index Terms*—#CSP; constraint networks; decomposition**

I. Introduction

Counting models in propositional logic (called #SAT) and counting solutions for constraint satisfaction problems (called #CSP) are challenging problems. They have numerous applications in AI, e.g. in approximate reasoning [1], in diagnosis [2], in belief revision [3], in probabilistic inference [4]–[7], in planning [8], [9], for guiding search in CSPs [10], and in other domains outside computer science as in statistical physics [11] or in chemistry for protein structure prediction [12].

However, these problems are extremely difficult from a theoretical point of view in terms of complexity because they are known as #P-complete [13]. Moreover we can claim that they are really hard considering Toda's theorem which shows that $PH \subseteq P^{\#P}$ [14]. On a practical level, their resolution is also very difficult. So, these problems have been studied for a long time and are the subject of numerous studies over last years. On the one hand, theoretical studies have been realized, trying to analyze these problems from the viewpoint of theoretical complexity by exhibiting tractable classes [15] or by analyzing their theoretical hardness by the means of dichotomy theorems [16]. On the other hand, from a practical perspective, solving methods have been proposed. Nevertheless, given the theoretical (and practical) hardness of these problems, most works try to solve the problem by approximating the solution, that is to say, by offering bounds of the number of solutions (or models). Indeed, it is often difficult or impossible to solve these problems accurately i.e. to obtain the exact number of solutions. So, most works have been achieved by sampling the search space [17]–[21]. All these methods except that in [17] provide a lower bound of the number of solutions with a high-confidence interval obtained by randomly assigning variables until solutions are found. A possible drawback of these approaches is that they might find no solution within a given time limit due to inconsistent partial assignments. For large and complex problems, this results in zero lower bounds or it requires time-consuming parameter (e.g. sample size) tuning in order to avoid this problem. Another approach involves reducing the search space by adding streamlining XOR constraints [22], [23]. However, the resulting problem is not necessarily easier to solve.

In contrast, by exploiting certain properties of instances, it is possible to provide exact methods that can be efficient w.r.t. theory and practice. We think especially about methods that exploit some features of the instances for which polynomial time algorithms may exist. This is the case for example when the constraint network representing the problem has tree-width bounded by a constant. Notably, in this paper, we are interested in search methods that exploit the problem structure, providing time and space complexity bounds like the d-DNNF compiler [24] and AND/OR graph search [21], [25]. In the same spirit, Favier et al. [26] have proposed to adapt Backtracking with Tree-Decomposition (BTD) [27] to #CSP. This method which is one of the most efficient structural decomposition methods for solving CSPs was initially proposed for solving structured instances. Their modifications to BTD, called #BTD, are similar to what has been done in the AND/OR context [21], [25], except that BTD relies on a tree-decomposition instead of a pseudo-tree. This naturally enables BTD to exploit dynamic variable orderings inside clusters whereas AND/OR search uses a static ordering. Even by exploiting such methods, it often remains difficult or impossible to solve these problems accurately. Nevertheless, it seems that the performances of such approaches can significantly be improved.

So, in this paper, we propose to improve the performances of the approach introduced in [26] and which is based on BTD [27], Here, the main objective is to offer better performance for the exact resolution. However, improving exact methods can also be useful to design better approximate methods.

Since the algorithm #BTD is based on tree-decomposition of constraints networks, it proceeds by solving independent parts of the network. More precisely, it explores systematically these independent parts of the search space to enumerate all their local solutions. This approach allows to define an algorithm whose implementation is relatively simple, and which ensures complexity bounds that are induced by the considered tree-decomposition. So, for constraint networks of bounded tree-width, it ensures to get a polynomial time algorithm. Unfortunately, from a practical viewpoint, such an approach can be really costly. Indeed, there is no guarantee that every local solution participates to at least a global solution (i.e. to a consistent assignment on all the variables of X). Thus, when a local solution cannot be consistently extended to the whole problem, the exhaustive search achieved by #BTD can cause a significant loss of time.

So, in this paper, we propose to improve the approach introduced in [26] offering a new version of #BTD called #EBTD, which is better suited to counting. When a new local solution is found, rather than looking for all its local extensions, #EBTD will look if in the whole problem, there is at least one global solution which is compatible with it. The idea may seem quite simple, but it requires significant changes in the algorithm #BTD. We describe this new algorithm and provide an analysis of its time and space complexities which demonstrates that the complexity bounds are preserved w.r.t. #BTD. To assess the practical advantages of #EBTD, we perform experiments on a large set of benchmarks. Compared to #BTD, this algorithm turns to be more efficient. But what is most interesting are the performances obtained when compared with methods of the state of the art. Indeed, our improvement induced a significant gain, both in terms of solved instances, but also in terms of computation time. Moreover, in the case of instances that cannot be completely solved by lack of time or memory, we show that #EBTD can often provide non-zero lower bounds of the number of solutions while, in such cases, #BTD and the other solvers we consider produce none.

This paper is organized as follows. In Section II, we introduce notations and recall tree-decomposition and the principles of #BTD. Section III describes #EBTD. Experimental results are given in Section IV. Then we conclude in Section V.

II. Preliminaries

A. Constraint Networks and #CSP

An instance of a finite *Constraint Satisfaction Problem* (CSP) is given by a triple (X, D, C), with $X = \{x_1, \dots, x_n\}$ a set of n variables, $D = (d_{x_1}, \dots, d_{x_n})$ a set of finite domains, and $C = \{c_1, \dots, c_e\}$ a set of e constraints. Each constraint c_i is a pair $(S(c_i), R(c_i))$, where $S(c_i) = \{x_{i_1}, \dots, x_{i_k}\} \subseteq X$ defines the *scope* of c_i, and $R(c_i) \subseteq d_{x_{i_1}} \times \dots \times d_{x_{i_k}}$ is its *compatibility relation*. The *arity* of c_i is $|S(c_i)|$. If the arity of each constraint is two, the instance is a *binary* CSP. The structure of a constraint network (other name of a CSP) is given by a hypergraph (a graph for a binary CSP), called the *constraint (hyper)graph*, whose vertices correspond to variables while edges correspond to the scopes of the constraints. To simplify notations, we denote the hypergraph $(X, \{S(c_1), \dots S(c_e)\})$ by (X, C). Without loss of generality, we assume that considered networks are connected. An assignment on a subset of X is called *consistent* if all the constraints are satisfied. Checking whether a CSP has a *solution* (a consistent assignment of X) is well known to be NP-complete. So, many works have been done to improve the solving in practice, even if the complexity of these approaches remains exponential, at least in $O(n.d^n)$ where d is the maximum size

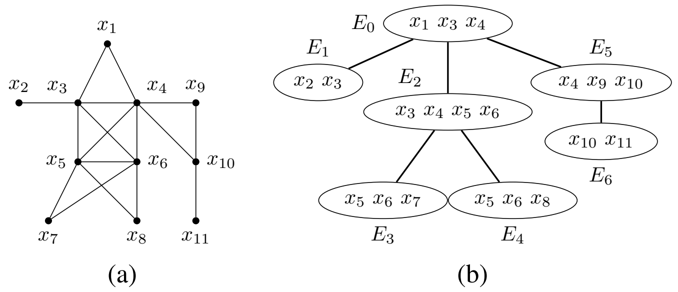


Fig. 1. A constraint graph (a) and one of its optimal tree-decompositions (b).

of domains. The problem of counting solutions for constraint satisfaction problem is called #CSP. This problem which is a generalization of #SAT is known to be #P-complete [13]. A simple way to solve it, consists in running algorithms as MAC [28] or RFL [29], but using an adaptation to enumerate all solutions. So, such algorithms run in $O(n.d^n)$.

B. Counting with Tree-Decomposition

To circumvent the theoretical intractabilities of CSP and #CSP, other approaches than the basic enumeration algorithms (with exponential time) have been proposed. Some of them rely on a structural tractable class based on the notion of *tree-decomposition of graphs* [30] (a tree of clusters of vertices):

Definition 1: A *tree-decomposition* of a graph $G = (X, C)$ is a pair (E, T) with $T = (I, F)$ a tree (I the set of nodes and F the set of edges of T) and $E = \{E_i : i \in I\}$ a family of subsets of X, such that each subset (called cluster) E_i is a node of T and satisfies: (i) $\cup_{i \in I} E_i = X$, (ii) for each edge $\{x, y\} \in C$, there exists $i \in I$ with $\{x, y\} \subseteq E_i$, and (iii) for all $i, j, k \in I$, if k is in a path from i to j in T, then $E_i \cap E_j \subseteq E_k$. The width of a tree-decomposition (E, T) is equal to $max_{i \in I} |E_i| - 1$. The *tree-width* w of G is the minimal width over all the tree-decompositions of G.

Note that this notion can also be considered when the network is a hypergraph using its *2-section*[1].

For example, Figure 1(b) presents a tree-decomposition of the graph in Figure 1(a). We get the clusters $E_0 = \{x_1, x_3, x_4\}$, $E_1 = \{x_2, x_3\}$, $E_2 = \{x_3, x_4, x_5, x_6\}$, $E_3 = \{x_5, x_6, x_7\}$, $E_4 = \{x_5, x_6, x_8\}$, $E_5 = \{x_4, x_9, x_{10}\}$ and $E_6 = \{x_{10}, x_{11}\}$. The cluster E_2 has the maximum size of 4. So the width of this decomposition is 3. Moreover, it is an optimal decomposition. So the tree-width w of this graph is 3. Given a root cluster E_r, we denote $Desc(E_j)$ the set of vertices (variables) belonging to the union of the descendants E_k of E_j in the tree rooted in E_j, E_j included. Let us consider that E_0 is the root cluster E_r. We then have $Desc(E_0) = X$ and $Desc(E_2) = \{x_3, x_4, x_5, x_6, x_7, x_8\}$.

The primary advantage of solving methods exploiting tree-decompositions is related to their theoretical complexity, d^{w+1} where w is the *tree-width* of the constraint network. Thus, these methods for solving CSP or #CSP instances can be efficient on large instances of small tree-width as it is the

[1]The 2-section of a hypergraph (X, C) is the graph (X, C') where $C' = \{\{x, y\} | \exists c \in C, \{x, y\} \subseteq c\}$ [31].

case for example for well known optimization problems of radio frequency allocations [32]. However, as computing the tree-width is an NP-hard task [33], the complexity of decomposition methods for solving CSPs is generally related to an approximation w^+ of w rather than the tree-width w itself. More precisely, the time complexity of such methods is $O(n.w^+.log(d).d^{w^++1})$ for a space complexity in $O(n.d^{w^++1})$, where $w^+ + 1$ is the size of the largest cluster ($w + 1 \leq w^+ + 1 \leq n$). This first approach called *Tree-Clustering* [34] has been improved to obtain a space complexity in $O(n.s.d^s)$ [27], [35] with s the size of the largest intersection (*separator*) between two clusters ($s \leq w^+$).

Now, we explain how the algorithm #BTD solves #CSP instances from a tree-decomposition of their constraint hypergraph. Using a tree-decomposition allows to exploit the structural properties of the constraint hypergraph. To do so, the considered tree-decomposition induces an order on the exploitation of the clusters. The order is said to be *compatible* if it can be produced by a depth-first traversal of T from the root cluster E_r. Given such a cluster ordering $<$, #BTD achieves a backtrack search by using a *compatible variable ordering* $\preceq$ s.t $\forall x \in E_i, \forall y \in E_j$, with $E_i < E_j$, $x \preceq y$. In other words, the cluster ordering induces a partial ordering on the variables since the variables in E_i are assigned before those in E_j if $E_i < E_j$. For the example of Figure 1, the cluster ordering $E_r = E_0 < E_1 < E_2 < E_3 < E_4 < E_5 < E_6$ is compatible for E and $x_1 \preceq x_3 \preceq x_4 \preceq x_2 \preceq x_5 \preceq x_6 \preceq x_7 \preceq x_8 \preceq x_9 \preceq x_{10} \preceq x_{11}$ is compatible for X. The essential property of tree decomposition is that assigning $E_i \cap E_j$ (E_j is a child cluster of E_i) separates the initial problem into two subproblems, which can then be solved independently. The first subproblem rooted in E_j is defined by the variables in $Desc(E_j)$ and by all the constraints involving *at least* one variable in $Desc(E_j) \setminus E_i$. In the following, the subproblem rooted on E_j and induced by a consistent assignment $\mathcal{A}$ on $E_i \cap E_j$ and some variables of $Desc(E_j)$ is denoted $P_{\mathcal{A},E_i/E_j}$. In $P_{\mathcal{A},E_i/E_j}$, all the variables assigned in $\mathcal{A}$ have a domain reduced to their assigned value in $\mathcal{A}$ while the other variables in $Desc(E_j)$ have their initial domain as domain. The second subproblem is formed by the remaining variables, together with the constraints involving them. #BTD can exploit this property by using any compatible variable ordering. So the variables of any cluster E_i will be assigned before the variables that remain in its children. In this case, for any cluster $E_j \in Children(E_i)$ (where $Children(E_i)$ denotes the set of children of the cluster E_i), once $E_i \cap E_j$ is assigned, the subproblem $P_{\mathcal{A},E_i/E_j}$ rooted in E_j and conditioned by the current assignment $\mathcal{A}$ of $E_i \cap E_j$ can be solved independently of the rest of the problem. The exact number of solutions $\#sol_{E_j}$ of this subproblem may then be recorded, as a *structural #good* ($\mathcal{A}$,$\#sol_{E_j}$). By this way, it will never be computed again for the same assignment of $E_i \cap E_j$. This is why algorithms such as BTD or #BTD (and also AND / OR graph search) are able to keep the complexity exponential in the size of the largest cluster only.

Algorithm 1: #BTD (P, (E,T), $\mathcal{A}$, E_i, V_{E_i}, G)

```
Input: A CSP $P = (X, D, C)$, a tree-decomposition $(E, T)$, the current
       assignment $\mathcal{A}$, the current cluster $E_i$, the set $V_{E_i}$ of unassigned variables
       in $E_i$
Input/output: the set $G$ of recorded #goods
Result: The number of solutions of $P_{\mathcal{A}[E_i \setminus V_{E_i}],E_{p(i)}/E_i}$
1  if $V_{E_i} = \emptyset$ then
2      $S_{E_i} \leftarrow Children(E_i)$
3      $\#sol \leftarrow 1$
4      while $S_{E_i} \neq \emptyset$ and $\#sol \neq 0$ do
5          Choose a cluster $E_j \in S_{E_i}$
6          $S_{E_i} \leftarrow S_{E_i} \setminus \{E_j\}$
7          if $(\mathcal{A}[E_j \cap E_j], \#sol_{E_j})$ is a #good in $G$ then
8              $\#sol \leftarrow \#sol \times \#sol_{E_j}$
9          else
10             $\#sol_{E_j} \leftarrow$ #BTD($\mathcal{A}$,$E_j$, $V_{E_i} \setminus (E_i \cap E_j)$,$G$)
11             Record $(\mathcal{A}[E_i \cap E_j], \#sol_{E_j})$ as #good of $E_i$ w.r.t. $E_j$ in $G$
12             $\#sol \leftarrow \#sol \times \#sol_{E_j}$
13     return $\#sol$
14 else
15     Choose $x \in V_{E_i}$
16     $d \leftarrow d_x$
17     $\#sol_x \leftarrow 0$
18     while $d \neq \emptyset$ do
19         Choose $v \in d$
20         $d \leftarrow d - \{v\}$
21         if $\mathcal{A} \cup \{x \leftarrow v\}$ satisfies all constraints of $C$ then
22             $\#sol_x \leftarrow \#sol_x +$#BTD($\mathcal{A} \cup \{x \leftarrow v\}$, $E_i$, $V_{E_i} \setminus \{x\}$,$G$)
23     return $\#sol_x$
```

#BTD is described in Algorithm 1. Given an assignment $\mathcal{A}$ and a cluster E_i, #BTD looks for the number of extensions $\mathcal{B}$ of $\mathcal{A}$ on $Desc(E_i)$ such that $\mathcal{A}[E_i \setminus V_{E_i}] = \mathcal{B}[E_i \setminus V_{E_i}]$. V_{E_i} denotes the set of unassigned variables of E_i while $A[Y]$ represents the restriction of the assignment $\mathcal{A}$ to the variables of Y. The first call is to #BTD($\emptyset, E_r, E_r$) and it returns the number of solutions of the whole problem. Inside a cluster E_i, #BTD proceeds classically by assigning a value to a variable and by backtracking if any constraint is violated (lines 15-22). When every variable in E_i is consistently assigned (line 1), #BTD computes the number of solutions of the subproblem induced by the first child of E_i, if there is one (lines 2-12). More generally, let us consider E_j, a child of E_i. Given a current assignment $\mathcal{A}$ on E_i, #BTD checks whether the assignment $\mathcal{A}[E_i \cap E_j]$ corresponds to a #good. If so, #BTD multiplies the recorded number of solutions with the number of solutions of E_i with $\mathcal{A}$ as its assignment. Otherwise, it extends $\mathcal{A}$ on $Desc(E_j)$ in order to compute its number of consistent extensions $\#sol_{E_j}$. Then, it records the #good $(\mathcal{A}[E_i \cap E_j], \#sol_{E_j})$. #BTD computes after the number of solutions of the subproblem induced by the next child of E_i. Finally, when each child of E_i has been examined, #BTD tries to modify the current assignment of E_i. The number of solutions of E_i is the sum of solution counts for every consistent assignment of E_i. The time (resp. space) complexity of #BTD for #CSP is the same as for CSP: $O(n.w^+.log(d).d^{w^++1})$ (resp. $O(n.s.d^s)$) [26].

In [26], the experiments have shown that in practice, and particularly for problems with large tree-width, #BTD runs frequently out of time and memory. Contrary to [26] which mainly used #BTD as a subroutine for an approximate method, we propose to improve directly this approach for exact counting. Indeed, we can see that given a partial assignment $\mathcal{A}$, a call to #BTD, such as in $\#sol_{E_j} \leftarrow$ #BTD($\mathcal{A}, E_j, V_{E_j} \backslash (E_i \cap E_j)$) (line 10), solves entirely the current subproblem, even if the partial assignment $\mathcal{A}$ cannot be extended to a solution on the whole problem. So, a first possible way to improve #BTD consists in avoiding such useless searches. In the next section, we exploit this idea in order to define a more sophisticated algorithm called #EBTD.

III. Improving Exact Counting with Tree-Decomposition

As #BTD, #EBTD (for *Enhanced Backtracking with Tree-Decomposition*) is based on the notion of tree-decomposition of graphs. #EBTD also achieves a backtrack search by using a compatible variable ordering. The first difference with #BTD is related to the concept of goods which is expanded:

Definition 2 (Exact and partial structural goods): Let (E, T) be a tree-decomposition, E_i and E_j two clusters from E where E_j is a child of E_i and $\mathcal{A}$ a consistent assignment on $E_i \cap E_j$. An *exact structural good* is a triplet ($\mathcal{A}$,=,$\#sol_{E_j}$) where $\#sol_{E_j}$ is the number of solutions of $P_{\mathcal{A},E_i/E_j}$. A *partial structural good* is a triplet ($\mathcal{A}$,$\geq$,$\#sol_{E_j}$) where $\#sol_{E_j}$ is a lower bound on the number of solutions of $P_{\mathcal{A},E_i/E_j}$.
Note that the exact structural goods ($\mathcal{A}$,=,$\#sol_{E_j}$) are identical to the structural #good($\mathcal{A}$,$\#sol_{E_j}$) defined for #BTD [26]. Moreover, we also exploit the notion of structural nogood:

Definition 3 (structural nogood [27]): Let (E, T) be a tree-decomposition and E_i and E_j be two clusters from E where E_j is a child of E_i. A *structural nogood* of E_i w.r.t. E_j is a consistent assignment $\mathcal{A}$ on the variables of $E_i \cap E_j$ such that $P_{\mathcal{A},E_i/E_j}$ has no solution.

#EBTD is described in Algorithm 2. Given a CSP $P = (X, D, C)$ and a tree-decomposition of (X, C), the call #EBTD (P, (E, T), $\mathcal{A}$, E_i, V_{E_i}, S, G, N) aims to compute the number of solutions of the subproblem $P_{\mathcal{A}[E_i \backslash V_{E_i}],E_{p(i)}/E_i}$ where $\mathcal{A}$ is the current assignment, E_i is the current cluster, $E_{p(i)}$ its parent cluster and V_{E_i} the set of unassigned variables of E_i. G and N represent respectively the set of structural goods and nogoods which have been recorded during the search. The algorithm exploits also a stack S which contains the next clusters to be explored. #EBTD returns a pair $(\#sol, E_{bt})$. E_{bt} represents the cluster to which #EBTD has to backtrack and allows to backjump to the relevant cluster when a nogood is found. Concerning the number of solutions $\#sol$, there are three possible cases:

- If $\#sol > 0$ and $E_{bt} = E_i$, then $\#sol$ is the exact number of solutions of the subproblem $P_{\mathcal{A}[E_i \backslash V_{E_i}],E_{p(i)}/E_i}$.
- If $\#sol > 0$ and $E_{bt} \neq E_i$, then $\#sol$ is a lower bound of the number of solutions of $P_{\mathcal{A}[E_i \backslash V_{E_i}],E_{p(i)}/E_i}$.
- If $\#sol = 0$ then $P_{\mathcal{A}[E_i \backslash V_{E_i}],E_{p(i)}/E_i}$ has no solution.

The resulting S can be defined as the set of clusters E_j such that $E_{p(j)}$ is assigned and $\mathcal{A}[E_{p(j)} \cap E_j]$ does not correspond to a partial or exact good of $E_{p(j)}$ w.r.t. E_j. We can note, that, it is necessarily empty in the first case.

The initial call is #EBTD $(P, (E, T), \mathcal{A}, E_r, E_r, S, G, N)$ where S, G and N are empty. Like #BTD, #EBTD, starts its backtrack search by assigning consistently the variables of the root cluster E_r before exploring a child cluster. When exploring a new cluster E_i, since the variables in the parent cluster $E_{p(i)}$ (and so in the separator $E_i \cap E_{p(i)}$[2]) are already assigned, it only has to assign the variables which appear in $E_i \backslash (E_i \cap E_{p(i)})$. In order to solve each cluster (lines 46-57), #EBTD (and #BTD) can exploit any solving algorithm which does not alter the structure. For sake of simplicity, the version presented in Algorithm 2 relies on Chronological Backtracking (BT). However, for the experiments provided in the next section, we will consider a more powerful algorithm, namely RFL (for Real Full Look-ahead [29]). Thus, #EBTD first chooses the next variable x to assign among the variables in V_{E_i} (line 46). Then, it selects a value v from d_x according to the value heuristic (lines 51-52) and assigns it to the variable x. If the new assignment is consistent w.r.t. the initial constraints of C and the structural nogoods of N (line 53), the search goes on by choosing a new variable and making a new assignment. Otherwise, #EBTD tries to assign a new possible value to the variable x. When all the possible values have been tried, #EBTD backtracks.

When #EBTD (like #BTD) has consistently assigned all the variables of the current cluster E_i, it then aims to count the number of solutions of each subproblem rooted in each child cluster E_j. More precisely, for a child E_j, it aims to count the number of solutions of $P_{\mathcal{A}[E_i \cap E_j],E_i/E_j}$. When this number $\#sol_{E_j}$ is computed, #EBTD records the exact good $(\mathcal{A}[E_i \cap E_j], =, \#sol_{E_j})$ (line 23). Similarly, #BTD records the corresponding #good $(\mathcal{A}[E_i \cap E_j], \#sol_{E_j})$ (line 11). The main difference between #EBTD and #BTD resides in the exploration of the children. #BTD computes the exact number of solutions of each subproblem $P_{\mathcal{A}[E_i \cap E_j],E_i/E_j}$ related to each child cluster E_j by considering successively the children. Thus, for the tree-decomposition of Figure 1(b), after assigning consistently the cluster E_0, #BTD computes the number of solutions of $P_{\mathcal{A}[E_0 \cap E_1],E_0/E_1}$, then one of $P_{\mathcal{A}[E_0 \cap E_2],E_0/E_2}$ and finally one of $P_{\mathcal{A}[E_0 \cap E_5],E_0/E_5}$. The disadvantage of such an approach is that all the solutions of $P_{\mathcal{A}[E_0 \cap E_1],E_0/E_1}$ are computed before even knowing if there is at least one solution for $P_{\mathcal{A}[E_0 \cap E_2],E_0/E_2}$ and more generally a global solution. In fact, in the worst case, #BTD may compute the number of solutions of $P_{\mathcal{A}[E_0 \cap E_1],E_0/E_1}$ and $P_{\mathcal{A}[E_0 \cap E_2],E_0/E_2}$ before establishing that $P_{\mathcal{A}[E_0 \cap E_5],E_0/E_5}$ has no solution. In this case, counting the number of solutions for the subproblem $P_{\mathcal{A}[E_i \cap E_j],E_i/E_j}$ related to a previous child cluster E_j is useless unless the recorded #good $(\mathcal{A}[E_i \cap E_j], \#sol_{E_j})$ is used later. In order to solve this issue, #EBTD proposes to explore the children in a different way. More precisely, it aims to guarantee that it counts exactly the number of solutions of the subproblem rooted on a child cluster E_j only if there is a global solution for the problem so that the recorded good $(\mathcal{A}[E_i \cap E_j], =, \#sol_{E_j})$ is necessarily used at least once. For

[2]We assume that $E_i \cap E_{p(i)} = \emptyset$ if E_i is the root cluster.

instance, if we consider again the tree-decomposition of Figure 1(b), after assigning consistently the cluster E_0, #EBTD tries to assign, by order, the clusters E_1, E_2, E_3, E_4, E_5 and E_6. Once the global solution is found, it computes then $\#sol_{E_5}$, $\#sol_{E_2}$ and finally $\#sol_{E_1}$. In contrast, if after assigning consistently $Desc(E_1)$ and $Desc(E_2)$, $Desc(E_5)$ cannot be consistently assigned, $\mathcal{A}[E_0 \cap E_5]$ is recorded as a nogood of E_0 w.r.t. E_5 while $\mathcal{A}[E_0 \cap E_2]$ and $\mathcal{A}[E_0 \cap E_1]$ are recorded as partial goods (resp. ($\mathcal{A}[E_0 \cap E_1]$,$\geq$,1) of E_0 w.r.t. E_1 and ($\mathcal{A}[E_0 \cap E_2]$,$\geq$,1) of E_0 w.r.t. E_2). These structural (no)goods can be used later to avoid exploring a redundant part of the search tree. The principle is applied recursively. Indeed, when counting $\#sol_{E_2}$, once the assignment on E_2 is changed #EBTD tries again to assign E_3 then E_4 (if needed since recorded goods may be used). Hence, the presence of a global solution is guaranteed before computing $\#sol_{E_4}$ and $\#sol_{E_3}$.

Lines 46-57 of Algorithm 2 explore the cluster E_i as $\#BT$ does. The condition $x \notin E_{bt}$ (line 58) suspends the enumeration of the solutions of E_i when a nogood of $E_{p(\ell)}$ w.r.t. E_ℓ is recorded where E_ℓ is a cluster explored after E_i. In this case, $E_{bt} = E_{p(\ell)}$ and the search backtracks to the cluster $E_{p(\ell)}$. When all the variables of E_i have been consistently assigned (line 1), #EBTD considers each child of E_i (lines 2-16). If $\mathcal{A}[E_i \cap E_j]$ corresponds to an exact good (line 10), the good is exploited and the search jumps to the next child cluster. #EBTD uses two locals stacks $S_{unknown}$ and $S_{good_\geq}$. It adds to S and $S_{unknown}$ each child E_j of E_i for which $\mathcal{A}[E_i \cap E_j]$ does not correspond to a good of E_i w.r.t. E_j. In the contrary, when $\mathcal{A}[E_i \cap E_j]$ corresponds to a partial good of E_i w.r.t. E_j, E_j is added to the $S_{good_\geq}$. $S_{good_\geq}$ is the set of children E_j of E_i for which $\mathcal{A}[E_i \cap E_j]$ corresponds to a partial good of E_i w.r.t. E_j with a lower bound of the number of solutions of the subproblem $P_{\mathcal{A}[E_i \cap E_j], E_i/E_j}$. For each cluster of $S_{good_\geq}$, the exact number of solutions of $P_{\mathcal{A}, E_i/E_j}$ is computed later (lines 38-43) if the current assignment $\mathcal{A}$ can be extended to a global solution. Also for each cluster of $S_{unknown}$, at line 36, the exact number of solutions of $P_{\mathcal{A}[E_i \cap E_j], E_i/E_j}$ is guaranteed to be computed (lines 17-37) and thus can be exploited. If the stack S is not empty (line 17), we keep on the search on the cluster E_j at the top of S (line 20). If #EBTD succeeds in extending $\mathcal{A}$ on the subproblem rooted in E_j, it records $\mathcal{A}[E_{p(j)} \cap E_j]$ as a good. If $\mathcal{A}$ can be extended to a global solution ($E_{bt} = E_j$), the number associated to this good is the number of solutions of $P_{\mathcal{A}[E_i \cap E_j], E_{p(j)}/E_j}$ and we have an exact good. Otherwise, it corresponds to a lower bound and we have a partial good. On the contrary, if $\mathcal{A}$ cannot be extended on the subproblem rooted in E_j, $\mathcal{A}[E_{p(j)} \cap E_j]$ is recorded as a nogood. Note that lines 27 and 33 allow to remove from the stack S the children of the cluster E_{bt} (if any) provided that a nogood of E_{bt} w.r.t. one of its children has been recorded.

Theorem 1: #EBTD is sound, complete and terminates.

Proof: Let $G_=(E_i)$ be the set of assignments corresponding to exact goods of $E_p(i)$ w.r.t. E_i in G. Let us consider $VAR(V_{E_i}, S, \mathcal{A}, G) =$

Algorithm 2: #EBTD (P, (E,T), $\mathcal{A}$, E_i, V_{E_i}, S, G, N)

```
Input: A CSP $P = (X, D, C)$, a tree-decomposition $(E, T)$, the current
       assignment $\mathcal{A}$, the current cluster $E_i$, the set $V_{E_i}$ of unassigned variables
       in $E_i$
Input/output: $S$ a stack of clusters, the set $G$ of recorded goods, the set $N$ of
       recorded nogoods
Result: (The number of solutions found for $P_{\mathcal{A}[E_i \setminus V_{E_i}], E_{p(i)}/E_i}$, the cluster
       in which we have to backtrack)
1  if $V_{E_i} = \emptyset$ then
2      $\#sol \leftarrow 1$
3      $S_{E_i} \leftarrow Children(E_i)$
4      $S_{good_\geq} \leftarrow \emptyset$
5      $S_{unknown} \leftarrow \emptyset$
6      while $S_{E_i} \neq \emptyset$ do
7          Choose a cluster $E_j \in S_{E_i}$
8          $S_{E_i} \leftarrow S_{E_i} \setminus \{E_j\}$
9          switch $\mathcal{A}[E_i \cap E_j]$ do
10             case good $(\mathcal{A}[E_i \cap E_j], =, \#sol_{E_j})$ of $E_i$ w.r.t. $E_j$ in $G$
11                 $\#sol \leftarrow \#sol * \#sol_{E_j}$
12             case good $(\mathcal{A}[E_i \cap E_j], \geq, \#sol_{E_j})$ of $E_i$ w.r.t. $E_j$ in $G$
13                 $S_{good_\geq} \leftarrow S_{good_\geq} \cup \{E_j\}$
14             otherwise
15                 $S_{unknown} \leftarrow S_{unknown} \cup \{E_j\}$
16                 $S \leftarrow S \cup \{E_j\}$
17     if $S \neq \emptyset$ then
18         $E_j \leftarrow Top(S)$
19         $S \leftarrow S \setminus \{E_j\}$
20         $(\#sol_{E_j}, E_{bt}) \leftarrow$ #EBTD($P$,$(E,T)$,$\mathcal{A}$,$E_j$,$E_j \setminus (E_{p(j)} \cap E_j)$,
           $S$,$G$,$N$)
21         if $\#sol_{E_j} > 0$ then
22             if $E_{bt} = E_j$ then
23                 Record $(\mathcal{A}[E_{p(j)} \cap E_j], =, \#sol_{E_j})$ as good of $E_{p(j)}$
                   w.r.t. $E_j$ in $G$
24             else
25                 Record $(\mathcal{A}[E_{p(j)} \cap E_j], \geq, \#sol_{E_j})$ as good of $E_{p(j)}$
                   w.r.t. $E_j$ in $G$
26                 if $E_i = E_{bt}$ then
27                     $S \leftarrow S \setminus Children(E_i)$
28                     return $(0, E_i)$
29                 else return $(\#sol, E_{bt})$
30         else
31             Record $\mathcal{A}[E_{p(j)} \cap E_j]$ as nogood of $E_{p(j)}$ w.r.t. $E_j$ in $N$
32             if $E_i = E_{p(j)}$ then
33                 $S \leftarrow S \setminus Children(E_i)$
34                 return $(0, E_i)$
35             else return $(\#sol, E_{p(j)})$
36     foreach $E_j \in S_{unknown}$ do
37         $\#sol \leftarrow \#sol * \#sol_{E_j}$
38     while $S_{good_\geq} \neq \emptyset$ do
39         Choose a cluster $E_j \in S_{good_\geq}$
40         $S_{good_\geq} \leftarrow S_{good_\geq} \setminus \{E_j\}$
41         $(\#sol_{E_j}, E_{Fail}) \leftarrow$ #EBTD($P$,$(E,T)$,$\mathcal{A}$,$E_j$,$E_j \setminus (E_i \cap E_j)$,
           $S$,$G$,$N$)
42         Record $(\mathcal{A}[E_i \cap E_j], =, \#sol_{E_j})$ as good of $E_i$ w.r.t. $E_j$ in $G$
43         $\#sol \leftarrow \#sol * \#sol_{E_j}$
44     return $(\#sol, E_i)$
45 else
46     Choose $x \in V_{E_i}$
47     $d \leftarrow d_x$
48     $\#sol_x \leftarrow 0$
49     $E_{bt} \leftarrow E_i$
50     repeat
51         Choose $v \in d$
52         $d \leftarrow d - \{v\}$
53         if $\mathcal{A} \cup \{x \leftarrow v\}$ satisfies all constraints of $C \cup N$ then
54             $(\#sol_{xv}, E_{bt}) \leftarrow$ #EBTD($P$,$(E,T)$,$\mathcal{A} \cup \{x \leftarrow v\}$,$E_i$,
               $V_{E_i} \setminus \{x\}$,$S$,$G$,$N$)
55             $\#sol_x \leftarrow \#sol_x + \#sol_{xv}$
56     until $d = \emptyset$ or $x \notin E_{bt}$
57     return $(\#sol_x, E_{bt})$
```

$V_{E_i} \cup (\bigcup_{E_j \in Children(E_i) | \mathcal{A}[E_j \cap E_i] \notin G_=(E_j)} Desc(E_j) \backslash (E_j \cap E_i)) \cup (\bigcup_{E_k \in S} Desc(E_k) \backslash (E_k \cap E_{p(k)}))$. $VAR(V_{E_i}, S, \mathcal{A}, G)$ corresponds to the set of variables which must be explored by #EBTD to determine if a global solution containing $\mathcal{A}$ exists and to count the exact number of solution of $P_{\mathcal{A}[E_i \backslash V_{E_i}], E_{p(i)}/E_i}$.

In order to prove the theorem, we prove by induction on $VAR(V_{E_i}, S, \mathcal{A}, G)$ the property $\mathcal{P}(VAR(V_{E_i}, S, \mathcal{A}, G))$ defined as: "#EBTD($\mathcal{A}, E_i, V_{E_i}, S, G, N$) returns a pair $(\#sol, E_{bt})$ where $\#sol$ is the exact number of solutions of $P_{\mathcal{A}[E_i \backslash V_{E_i}], E_{p(i)}/E_i}$ if $E_i = E_{bt}$, a lower bound otherwise".

For lack of space, we will simply highlight the main ideas of the proof. When there is no ambiguity, $VAR(V_{E_i}, S, \mathcal{A}, G)$ is denoted VAR.

- Basis: Consider $\mathcal{P}(\emptyset)$. If $VAR = \emptyset$ then $V_{E_i} = \emptyset$ and $S = \emptyset$. Also, for each child E_j of E_i, $\mathcal{A}[E_j \cap E_i] \in G_=(E_j)$. Thus, by considering each child E_j of E_i (lines 10-11), #EBTD updates successively the number of solutions $\#sol$. Since for each child E_j of E_i $\mathcal{A}[E_j \cap E_i]$ is an exact good, $S_{good_\geq}$, $S_{unknown}$ and S remain empty. Hence, #EBTD returns $(\#sol, E_i)$ where $\#sol$ is the exact number of solutions of $P_{\mathcal{A}[E_i], E_{p(i)}/E_i}$. If E_i is a leaf cluster, #EBTD returns $(1, E_i)$ since the only possible extension of $\mathcal{A}$ is $\mathcal{A}$. So the property holds.
- Inductive step: Consider now $\mathcal{P}(VAR)$ with $VAR \neq \emptyset$. Assume that $\forall VAR' \subset VAR$, $\mathcal{P}(VAR')$ holds.
  If $V_{E_i} \neq \emptyset$: the loop (lines 50-56) explores the cluster E_i as #BT does. The only difference is that the loop is suspended when $E_{bt} \neq E_i$ which is helpful when a nogood is found later during the search. If $E_{bt} = E_i$, the loop is equivalent to the loop of #BT and $\#sol_x$ is the number of solutions of $P_{\mathcal{A}[E_i \backslash V_{E_i}], E_{p(i)}/E_i}$ for the values v already assigned to x. When #EBTD attempts to assign v to x, if $\mathcal{A} \cup \{x \leftarrow v\}$ is inconsistent, there is no possible extension and $\#sol_x$ and E_{bt} are unchanged. In contrast, if $\mathcal{A} \cup \{x \leftarrow v\}$ is consistent, we call #EBTD (line 54) on $V_{E_i} \backslash \{x\}$. So, from the hypothesis of induction, we know that $\mathcal{P}(VAR \backslash \{x\})$ holds. If $E_{bt} = E_i$ then $\#sol_{xv}$ is the exact number of solutions and $\#sol_x$ is updated. So, the statement is valid. When $E_{bt} \neq E_i$, $\#sol_{xv}$ is a lower bound on the number of solutions and $\#sol_x$ is updated. Since x is assigned for the first time in E_i then $x \notin E_{bt}$ and the loop is suspended returning a lower bound on $P_{\mathcal{A}[E_i \backslash V_{E_i}], E_{p(i)}/E_i}$. Thus, the property holds.
  If $V_{E_i} = \emptyset$: Since $VAR \neq \emptyset$, for each child E_j of E_i, $\mathcal{A}[E_j \cap E_i]$ may be a good or an unknown assignment. Also, S can be empty or not. Lines 6-16 consider each child E_j and update, if needed, $\#sol$, $S_{good_\geq}$, $S_{unknown}$ and S. Lines 17-35 permit to launch the exploration of the clusters of S in order to determine if a global solution containing $\mathcal{A}$ exists. At line 36, a global solution is already found and for each child $E_j \in S_{unknown}$, $\#sol_{E_j}$ has been computed and $\#sol$ is updated (line 37). If $S_{good_\geq} \neq \emptyset$, each cluster $E_j \in S_{good_\geq}$ is explored (line 41). If $VAR(V_{E_j}, S, \mathcal{A}, G) \subset VAR(V_{E_i}, S, \mathcal{A}, G)$, by the induction hypothesis, $\mathcal{P}(VAR(V_{E_j}, S, \mathcal{A}, G))$ holds. Otherwise, $VAR(V_{E_j}, S, \mathcal{A}, G) = VAR(V_{E_i}, S, \mathcal{A}, G)$. But we know that for the next call, $V_{E_j} \neq \emptyset$. As shown above (case $V_{E_i} \neq \emptyset$), the property $\mathcal{P}(VAR(V_{E_j}, S, \mathcal{A}, G))$ holds. By definition of a partial good, $P_{\mathcal{A}[E_i \backslash V_{E_j}], E_i/E_j}$ has at least one solution. Since S is empty (a global solution is found), the recursive call of #EBTD returns necessarily the exact number of solutions of $P_{\mathcal{A}[E_j \backslash V_{E_j}], E_i/E_j} = P_{\mathcal{A}[E_i \backslash (E_i \cap E_j)], E_i/E_j}$. After all the children E_j are considered, $\#sol$ is updated and #EBTD returns $(\#sol, E_i)$. Hence, the property holds. If at line 17, $S \neq \emptyset$, a global solution containing $\mathcal{A}$ is not found yet. For the call of #EBTD at line 20, we can show, like previously for the call of line 41, that $\mathcal{P}(VAR(V_{E_j}, S, \mathcal{A}, G))$ holds. So $\mathcal{P}(VAR(V_{E_i}, S, \mathcal{A}, G))$ holds. □

Finally, we give its time and space complexities which are comparable to ones of #BTD.

Theorem 2: #EBTD has a time complexity in $O(n.(re + ns).d^{w^+ + 1})$ and a space complexity in $O(n.s.d^s)$ with $r = max_{c \in C} |S(c)|$.

Proof: The maximum number of constraints involved in each consistency check is e and the cost of a constraint check is $O(r)$. Likewise, if we assume that the nogoods of a cluster E_i w.r.t. to one of its children E_j are stored as a constraint whose scope is $E_i \cap E_j$, there are at most n-1 separators (since the number of clusters is bounded by n) and checking the existence of a nogood can be done in $O(s)$. Hence the consistency check of line 53 is achieved in $O(re + ns)$.

In the worst case, #EBTD explores all the clusters of the decomposition and tries all possible values of each variable of the cluster. Since w^+ is the width of the tree-decomposition, the maximum size of a cluster is $w^+ + 1$. Hence, the number of assignments of a cluster is bounded by $d^{w^+ + 1}$. A recorded exact good ensures that the corresponding cluster is not explored more than once with the same assignment on the variables of its separator. If a partial good is recorded, the cluster is, at most, visited twice with the same assignment so that the second time the cluster is fully explored and the partial good is replaced by an exact good. Moreover, assuming that the goods are stored on each separator like nogoods, storing a good or checking its existence can be done in $O(s)$. Consequently, #EBTD has a time complexity in $O(n.(re + ns).d^{w^+ + 1})$.

Concerning the space complexity, #EBTD records exact and partial goods and nogoods. These recordings are done w.r.t. a separator $E_i \cap E_j$ where E_j is a child of E_i. Provided that s is the maximum size of the separators, the size of a no(good) is bounded by s. For each separator, there are at most d^s possible assignments. So, as the number of separators is bounded by n, the space complexity is in $O(n.s.d^s)$. □

Note that, when #EBTD explores, for the second time, a given subproblem in order to compute an exact good, we can avoid restarting from scratch. Indeed, if we assume that we use a lexicographical value ordering, it suffices to record the partial solution related to the partial good conjointly with this partial good. By so doing, if #EBTD needs to explore again the corresponding subproblem, it can safely restart from the partial solution. Such an issue does not change the time complexity while the space complexity is now in $O(n.w^+.d^s)$. However, from a practical viewpoint, the gains may be significant.

IV. Experiments

In this section, we assess the practical interest of our approach. We consider here a version of #EBTD based on RFL [29] and we implement it in C++ in our own library. We exploit the $Min\text{-}Fill$ heuristic [36] in order to compute tree-decompositions. The arc-consistency is enforced by $AC3^{rm}$ for the preprocessing and $AC8^{rm}$ for the solving [37]. The next variable to assign is chosen thanks to the heuristic dom/wdeg [38] while the root cluster is a cluster maximizing the number of constraints whose scope intersects the cluster. The experiments are performed on blade servers running Linux Ubuntu 14.04 each with two Intel Xeon processors E5-2609 v2 2.5 GHz and 32 GB of memory. For each instance, the solving is performed within a timeout of 20 minutes and with at most 16 GB of memory.

Initially, we consider a first benchmark set B_1 containing 2,298 consistent CSP instances from the CSP 2008 competition[3]. Regarding the instances selection, we have excluded the instances having a trivial tree-decomposition (e.g. instances having a complete constraint graph) and the instances having global constraints (because global constraints are not taken into account yet by our CSP library). Among these 2,298 instances, #EBTD succeeds in counting exactly the number of solutions for 1,277 instances. For the other instances, it runs out of memory or reaches the timeout. However, it is still able to provide a non-zero lower bound of the number of solutions for 883 instances. At the end, for about 94% of the considered instances, #EBTD computes either the number of solutions or a non-zero lower bound.

Now we compare our approach with solvers from the state of the art. For #CSP solvers, we consider the implementation of #BTD provided in Toulbar2 [26]. Regarding #SAT solvers, we take into account Cachet [39], c2d [24], relsat [40] and sharpSAT [41]. In both cases, we have to encode the considered instances into a format supported by these solvers. For #BTD, we encode the CSP instances into the WCSP format while for #SAT solvers, we exploit the direct encoding from CSP to SAT [42]. These two encodings require that the constraints are flattened. Such an operation may need a large amount of time and memory for the instances having predicate constraints. So, in our comparisons, we exploit the benchmark set B_2 built from B_1 by considering only the 1,199 instances having no predicate constraint. For B_2, #EBTD succeeds in counting exactly the number of solutions for 1,006 instances against 976 for #BTD. Unfortunately, when comparing with #SAT solvers, we have to consider a benchmark subset B_3 from B_2 containing 1,059 instances because for 140 instances, the direct encoding from CSP to SAT is too time and space expensive. For B_3, #EBTD solves the largest number of instances (namely 908 instances) and is followed by sharpSAT (899 instances) and #BTD (877 instances). Cachet and relsat are less efficient (resp. with 608 and 618 instances) while c2d only solves 382 instances. Hence, in the following, we focus our comparison on #EBTD, #BTD and sharpSAT.

In Figure 2, we compare the runtime of #EBTD with one of #BTD (a) and sharpSAT (b) for the 840 instances of B_3 solved by the three solvers. Note that for #BTD and sharpSAT, the runtime does not include the encoding time. Clearly, #EBTD performs faster than #BTD for most of the instances. Moreover, the cumulative runtime of #EBTD for all these instances is 54,590 s while #BTD requires 98,373 s. The comparison between #EBTD and sharpSAT turns to be more balanced. Indeed, #EBTD is more efficient than sharpSAT for 63% of the instances (against 82% w.r.t. #BTD). However, globally, sharpSAT performs slower than #EBTD with a cumulative runtime of 93,284 s.

Finally, note that except #EBTD, none of the considered solvers provides a lower bound when the solving runs out of time or memory.

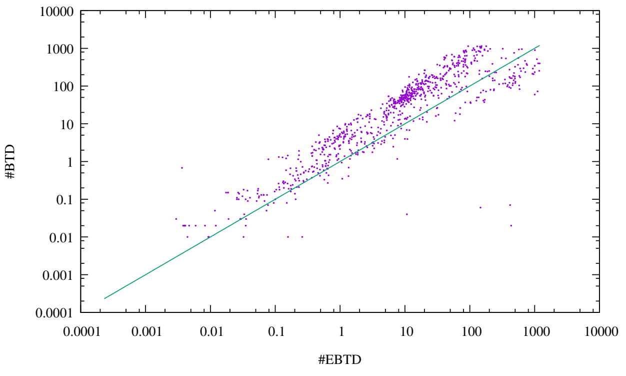
V. Conclusion

In this paper, we have proposed a new algorithm, called #EBTD, which is dedicated to solving the exact counting problem for CSPs, i.e. #CSP. This algorithm is based on constraint network decompositions and improves the previous algorithm introduced in [26]. #EBTD ensures non trivial complexity bounds for time and space which are related to structural properties of the considered constraint network. While it is based on a quite simple idea, its implementation requires a significant modification of the basic algorithm of [26]. For example, it requires the definition of the concept of *partial structural goods* whose use allows to avoid to explore large parts of the search space. Such an approach, of course, does not allow to improve the theoretical complexity, but it leads to a considerable improvement in computation time. Indeed, the experiments we conducted show that this new approach solves more instances on different classes of benchmarks. Above all, the solving is generally faster than the approaches of the state of the art, like for example sharpSAT.

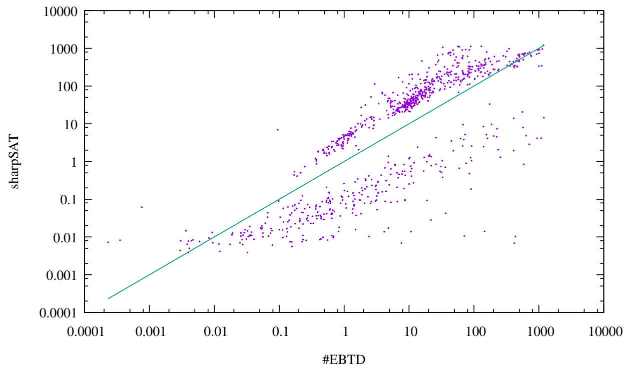
Several extensions of this work are possible. Among them, in the spirit of [43], it would be useful to study decompositions more suited to the counting problem. Indeed, according to the nature of the problem addressed (decision, optimization or counting), it is possible that some classes of decompositions are better suited for solving #CSP.

Moreover, for instances of large tree-width, for which exact counting is too hard, we should study how the use of #EBTD may provide better approximations. Finally, as many applications of the counting problem coming from AI are naturally expressed in propositional logic, it would be interesting to adapt #EBTD to Boolean instance, in order to improve directly its practical efficiency rather than using translation of instances from logic to CSPs.

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



(a)



(b)

Fig. 2. Runtime comparisons for #EBTD and #BTD (a) or sharpSAT (b) on the 840 instances of B_3 solved by the three solvers.

Acknowledgments

We thank Christian Bessière who has recently informed us of the existence of a similar work in Karakashian's thesis [44].

References

[1] D. Roth, "On the hardness of approximate reasonning," *Artificial Intelligence*, vol. 82, pp. 273–302, 1996.

[2] T. S. Kumar, "A model counting characterization of diagnoses," in *International Workshop on Principles of Diagnosis*, 2002.

[3] A. Darwiche, "On the tractable counting of theory models and its applications to truth maintenance and belief revision," *Journal of Applied Non-classical Logic*, vol. 11, pp. 11–34, 2001.

[4] T. Sang, P. Beame, and H. A. Kautz, "Performing Bayesian inference by weighted model counting," in *AAAI*, 2005, pp. 475–482.

[5] M. Chavira and A. Darwiche, "On probabilistic inference by weighted model counting," *AIJ*, vol. 172, pp. 772–799, 2008.

[6] U. Apsel and R. I. Brafman, "Lifted MEU by weighted model counting," in *AAAI*, 2005, pp. 1861–1867.

[7] A. Choi, D. Kisa, and A. Darwiche, "Compiling probabilistic graphical models using sentential decision diagrams," in *ECSQARU*, 2013, pp. 121–132.

[8] H. Palacios, B. Bonet, A. Darwiche, and H. Geffner, "Pruning conformant plans by counting models on compiled d-DNNF representations," in *ICAPS*, 2005, pp. 141–150.

[9] C. Domshlak and J. Hoffmann, "Fast probabilistic planning through weighted model counting," in *ICAPS*, 2006, pp. 243–252.

[10] K. Kask, R. Dechter, and V. Gogate, "Counting-Based Look-Ahead Schemes for Constraint Satisfaction," in *CP*, 2004, pp. 317–331.

[11] R. Burton and J. Steif, "Nonuniqueness of measures of maximal entropy for subshifts of finite type," *Ergodic theory and dynamical system*, vol. 14, pp. 213–235, 1994.

[12] M. Mann, G. Tack, and S. Will, "Decomposition During Search for Propagation-Based Constraint Solvers," *CoRR abs/0712.2389*, 2007.

[13] L. Valiant, "The Complexity of Computing the Permanent," *Theoretical Computer Science*, vol. 8, pp. 189–201, 1979.

[14] S. Toda, "PP is as hard as the polynomial-time hierarchy," *SIAM Journal on Computing*, vol. 20, pp. 865–877, 1991.

[15] F. Slivovsky and S. Szeider, "Counting Model for Formulas of Bounded Clique-Width," in *ISAAC*, 2013, pp. 677–687.

[16] A. Bulatov and V. Dalmau, "Towards a dichotomy theorem for the counting constraint satisfaction problem," *Information and Computation*, vol. 205, pp. 651–678, 2007.

[17] W. Wei and B. Selman, "A New Approach to Model Counting," in *SAT*, 2005, pp. 324–339.

[18] C. Gomes, J. Hoffmann, A. Sabharwal, and B. Selman, "From sampling to model counting," in *IJCAI*, 2007, pp. 2293–2299.

[19] V. Gogate and R. Dechter, "Approximate counting by sampling the backtrack-free search space," in *AAAI*, 2007, pp. 198–203.

[20] L. Kroc, A. Sabharwal, and B. Selman, "Leveraging belief propagation, backtrack search, and statistics for model counting," in *CPAIOR*, 2008, pp. 127–141.

[21] V. Gogate and R. Dechter, "Approximate Solution Sampling (and Counting) on AND/OR search spaces," in *CP*, 2008, pp. 534–538.

[22] C. Gomes, A. Sabharwal, and B. Selman, "Model counting: A new strategy for obtaining good bounds," in *AAAI*, 2006, pp. 54–61.

[23] C. Gomes, W.-J. van Hoeve, A. Sabharwal, and B. Selman, "Counting CSP solutions using generalized XOR constraints," in *AAAI*, 2007, pp. 204–209.

[24] A. Darwiche, "New Advances in Compiling CNF into Decomposable Negation Normal Form," in *ECAI*, 2004, pp. 328–332.

[25] R. Dechter and R. Mateescu, "The Impact of AND/OR Search Spaces on Constraint Satisfaction and Counting," in *CP*, 2004, pp. 731–736.

[26] A. Favier, S. de Givry, and P. Jégou, "Exploiting Problem Structure for Solution Counting," in *CP*, 2009, pp. 335–343.

[27] P. Jégou and C. Terrioux, "Hybrid backtracking bounded by tree-decomposition of constraint networks," *AIJ*, vol. 146, pp. 43–75, 2003.

[28] D. Sabin and E. Freuder, "Contradicting Conventional Wisdom in Constraint Satisfaction," in *ECAI*, 1994, pp. 125–129.

[29] B. Nadel, *Tree Search and Arc Consistency in Constraint-Satisfaction Algorithms*, ser. *Search in Artificial Intelligence*. Springer-Verlag, 1988, pp. 287–342.

[30] N. Robertson and P. Seymour, "Graph minors II: Algorithmic aspects of treewidth," *Algorithms*, vol. 7, pp. 309–322, 1986.

[31] C. Berge, *Graphs and Hypergraphs*. Elsevier, 1973.

[32] C. Cabon, S. de Givry, L. Lobjois, T. Schiex, and J. P. Warners, "Radio Link Frequency Assignment," *Constraints*, vol. 4, pp. 79–89, 1999.

[33] S. Arnborg, D. Corneil, and A. Proskuroswki, "Complexity of finding embeddings in a k-tree," *SIAM Journal of Disc. Math.*, vol. 8, pp. 277–284, 1987.

[34] R. Dechter and J. Pearl, "Tree-Clustering for Constraint Networks," *Artificial Intelligence*, vol. 38, pp. 353–366, 1989.

[35] R. Dechter and Y. E. Fattah, "Topological Parameters for Time-Space Tradeoff," *Artificial Intelligence*, vol. 125, pp. 93–118, 2001.

[36] D. J. Rose, "A graph theoretic study of the numerical solution of sparse positive definite systems of linear equations," in *Graph Theory and Computing*. Academic Press, 1972, pp. 183–217.

[37] C. Lecoutre, C. Likitvivatanavong, S. Shannon, R. Yap, and Y. Zhang., "Maintaining Arc Consistency with Multiple Residues," *Constraint Programming Letters*, vol. 2, pp. 3–19, 2008.

[38] F. Boussemart, F. Hemery, C. Lecoutre, and L. Sais, "Boosting systematic search by weighting constraints," in *ECAI*, 2004, pp. 146–150.

[39] T. Sang, F. Bacchus, P. Beame, H. Kautz, and T. Pitassi, "Combining Component Caching and Clause Learning for Effective Model Counting," in *SAT*, 2004.

[40] R. Bayardo and J. Pehoushek, "Counting Models Using Connected Components," in *AAAI*, 2000, pp. 157–162.

[41] M. Thurley, "sharpSAT - Counting Models with Advanced Component Caching and Implicit BCP," in *SAT*, 2006, pp. 424–429.

[42] T. Walsh, "SAT v CSP," in *CP*, 2000, pp. 441–456.

[43] P. Jégou, H. Kanso, and C. Terrioux, "An Algorithmic Framework for Decomposing Constraint Networks." in *ICTAI*, 2015, pp. 1–8.

[44] S. Karakashian, "Practical Tractability of CSPs by Higher Level Consistency and Tree Decomposition," Ph.D. dissertation, University of Nebraska-Lincoln, USA, 2013.