UDC 519.157

A. Favier, S. de Givry, Ph. Jégou

Solution Counting for CSP and SAT with Large Tree-Width

Рассмотрена проблема подсчета количества решений задачи совместимости ограничений (*Constraint Satisfaction Problem*). Для ее решения был адаптирован метод обратного прослеживания с ацикличным представлением графа ограничений (*Backtracking with Tree-Decomposition*). Предложен точный алгоритм, сложность которого экспоненциально зависит от ширины дерева, и приближенный алгоритм, экспоненциально зависящий от размера максимальной клики.

The problem of counting the number of solutions of a CSP is considered. For solving the problem the Backtracking with a Tree-Decomposition method was adapted. The exact algorithm is suggested which has the worst-time complexity exponential in a tree width, as well as iterative algorithm that has computational complexity exponential in a maximum clique size.

Розглянуто проблему підрахунку кількості розв'язків задачі сумісності обмежень (*Constraint Satisfaction Problem*). Для її розв'язку було адаптовано метод зворотного простеження з ациклічним поданням графа обмежень (*Backtracking with Tree-Decomposition*). Запропоновано точний алгоритм, складність якого експоненційно залежить від ширини дерева, і наближений алгоритм, експоненційно залежний від розміру максимальної кліки.

Abstract

This paper deals with the challenging problem of counting the number of solutions of a CSP, denoted #CSP. Recent progress has been made using search methods, such as *Backtracking with Tree-Decomposition* (BTD) [Jégou and Terrioux, 2003], which exploit the constraint graph structure in order to solve CSPs. We propose to adapt BTD for solving the #CSP problem. The resulting exact counting method has a worst-case time complexity exponential in a specific graph parameter, called *tree-width*. For problems with a sparse constraint graph but a large tree-width, we propose an iterative method which approximates the number of solutions by solving a partition of the set of constraints into a collection of partial chordal subgraphs. Its time complexity is exponential in the maximum clique size – the *clique number* – of the original problem, which can be much smaller than its tree-width. Experiments on CSP and SAT benchmarks show the practical efficiency of our proposed approaches[1].

1. Introduction

The Constraint Satisfaction Problem (CSP) formalism offers a powerful framework for representing and solving efficiently many problems. Finding a solution is NP-complete. A more difficult problem consists in counting the number of solutions. This problem, denoted #CSP, is known to be #P-complete [Valiant, 1979]. This problem has numerous applications in computer science, particularly in AI, *e.g.* in approximate reasoning [Roth, 1996], in belief revision [Darwiche, 2001], in diagnosis [Kumar, 2002], in guiding backtrack search heuristics to find solutions to CSPs [Zanarini and Pesant, 2009], and in other domains outside computer science such as in statistical physics [Burton and Steif, 1994] or in computational biology for protein structure prediction [Mann et al., 2007].

In the literature, two principal classes of approaches have been proposed. In the first class, methods find exactly the number of solutions in exponential time. In the second class, methods give approximations in a reasonable time. For the first class, a natural approach consists in extending classical search algorithms such as FC or MAC in order to enumerate all solutions. But the more solutions there are, the longer it takes to enumerate them.

Here, we are interested in search methods that exploit the problem structure, providing time and space complexity bounds. This is the case for the d-DNNF compiler `c2d` [Darwiche, 2004] and AND/OR graph search [Dechter and Mateescu, 2004, 2007] for counting. We propose to adapt Backtracking with Tree-Decomposition (BTD) [Jégou and Terrioux, 2003] to #CSP. This method was initially proposed for solving structured CSPs. Our modifications to BTD, resulting in an

Keywords: solution counting, constraint satisfaction problem, dynamic programming, approximate counting, chordal graph, graph coloring.

[1]A preliminary version appears in [Favier et al., 2009].

algorithm called #BTD, are similar to what has been done in the AND/OR context [Dechter and Mateescu, 2004, 2007], except that #BTD is based on a cluster tree-decomposition instead of a pseudo-tree, which naturally enables #BTD to exploit dynamic variable orderings inside clusters whereas AND/OR search uses a static ordering.

Most of the recent work on counting has been realized on a specific case of #CSP called #SAT, the model counting problem associated with SAT [Valiant, 1979]. Exact methods for #SAT extend systematic SAT solvers, adding component analysis [Bayardo and Pehoushek, 2000] (Relsat solver) and caching [Sang et al., 2004] (Cachet, further improved by sharpSAT [Thurley, 2006]) for efficiency purposes.

Approaches using approximations estimate the number of solutions. They propose poly-time or exponential time algorithms which should offer reasonably good approximations of the number of solutions, with theoretical guarantees about the quality of the approximation, or not. Most of the work has been done by sampling either the original OR search space [Wei and Selman, 2005, Gomes et al., 2007a, Gogate and Dechter, 2007, Kroc et al., 2008], or the original AND/OR search space [Gogate and Dechter, 2008]. All these methods except that in [Gogate and Dechter, 2008] provide a lower bound on 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 solution is to rely on a complete search method, withdrawing any time limit, in order to check whether every variable assignment made during the sampling process is globally consistent or not and then backtrack as done in [Gogate and Dechter, 2007].

Another approach involves reducing the search space by adding streamlining XOR constraints [Gomes et al., 2006, 2007b]. However, it does not guarantee that the resulting problem is easier to solve. A good overview of state-of-the-art exact and approximate counting methods for #SAT is given in [Gomes et al., 2009].

In this paper, we propose to relax the problem, by partitioning the set of constraints into a collection of structured chordal subproblems. Each subproblem is then solved using #BTD. Finally, an approximate number of solutions on the whole problem is obtained by combining the results of each subproblem. The resulting approximate method is called `Approx#BTD`. The task of counting the number of solutions of each subproblem should be relatively easy if the original instance has a sparse graph. In fact, it depends on the tree-width of the subproblems, which is bounded by the maximum clique size of the original instance called the *clique number*. In the case of a sparse graph, we expect this number to be small. This also forbids using our approach for CSPs with global constraints (*i.e.* having a complete constraint graph) or propositional CNF formulae with very large clauses. `Approx#BTD` gives also a trivial upper bound on the number of solutions.

Other relaxation-based counting methods have been tried in the literature such as mini-bucket elimination and iterative join-graph propagation [Kask et al., 2004], or in the related context of Bayesian inference, iterative belief propagation and the edge deletion framework [Choi and Darwiche, 2006][2]. These approaches do not exploit the local structure of the instances as it is done by search methods such as #BTD , thanks to local consistency and dynamic variable ordering.

In the next section, we introduce notation and the notion of a tree-decomposition. Section 3 describes #BTD for exact counting and Section 4 presents `Approx#BTD` for approximate counting. Experimental results are given in Section 5, then we conclude.

2. Preliminaries

A *Constraint Satisfaction Problem* [Montanari, 1974] is a quadruplet $\mathcal{P} = (X, D, C, R)$. X is a set

[2]It starts by solving an initial polytree-structured subproblem, further augmented by progressively recovering some edges, until the whole problem is solved. Approx#BTD starts directly with a possibly larger chordal subproblem.

of n variables with finite domains D. The domain of variable $x_i \in X$ with $i \in [1,n]$ is denoted $d_{x_i} \in D$. The maximum domain size is $d = \max_{i\in[1,n]} |d_{x_i}|$. C is a set of m constraints. Each constraint $c \in C$ is a set $\{x_{c_1},\ldots,x_{c_k}\} \subseteq X$ of variables. The problem is called a *binary* CSP if all the constraints have $k \le 2$. A relation $r_c \in R$ is associated with each constraint c such that r_c represents the set of allowed tuples over $d_{x_{c_1}} \times\cdots\times d_{x_{c_k}}$ for the assignment of the variables in c. Note that we can also define constraints by using functions or predicates for instance. An *assignment* of $Y = \{x_1,\ldots,x_k\} \subseteq X$ is a tuple $\mathcal{A} = (v_1,\ldots,v_k)$ from $d_{x_1} \times\cdots\times d_{x_k}$. We note the assignment $(v_1,\ldots,v_k)$ in the more meaningful form $(x_1 \leftarrow v_1,\ldots,x_k \leftarrow v_k)$. The projection of a tuple $\mathcal{A}$ over a subset of variables $c \subseteq Y$ is denoted $\mathcal{A}[c]$. A constraint c is said *satisfied* by $\mathcal{A}$ if $c \subseteq Y$ and $\mathcal{A}[c] \in r_c$, *violated* otherwise. $\mathcal{A}$ is said *consistent* with respect to a given subproblem if it satisfies all its constraints. A solution is an assignment of all the variables satisfying all the constraints.

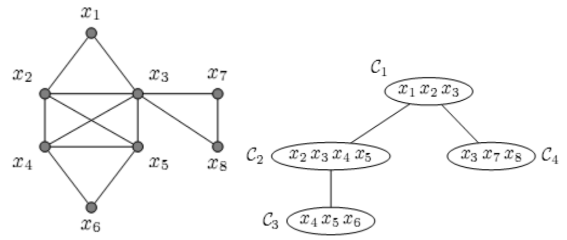
The structure of a CSP can be represented by the graph (X, C), called the *constraint graph*, whose vertices are the variables of X and with an edge between two vertices if the corresponding variables share a constraint in C. A graph is *chordal* if every cycle of length at least four has a chord, *i.e.* an edge joining two non-consecutive vertices along the cycle.

A *tree-decomposition* [Robertson and Seymour, 1986] of a CSP $\mathcal{P}$ is a pair $(\mathcal{C},\mathcal{T})$ with $\mathcal{T} = (I,F)$ a tree with vertices I and edges F and $\mathcal{C} = \{\mathcal{C}_i : i \in I\}$ a family of subsets of X, such that each cluster $\mathcal{C}_i$ is a node of $\mathcal{T}$ and satisfies: (1) $\cup_{i\in I} \mathcal{C}_i = X$, (2) for each constraint $c \in C$, there exists $i \in I$ with $c \subseteq \mathcal{C}_i$, (3) for all $i, j, k \in I$, if k is on a path from i to j in $\mathcal{T}$, then $\mathcal{C}_i \cap \mathcal{C}_j \subseteq \mathcal{C}_k$. The width of a tree-decomposition $(\mathcal{C},\mathcal{T})$ is equal to $\max_{i\in I} |\mathcal{C}_i| - 1$. The *tree-width* of $\mathcal{P}$ is the minimum width over all its tree-decompositions. Finding an optimal tree-decomposition is NP-Hard [Arnborg et al., 1987].

A tree-decomposition can be found by triangulation of (*i.e.* adding edges to) the constraint graph such that it becomes chordal and then by searching the maximal cliques of the triangulated constraint graph (resulting in the clusters $\mathcal{C}$) and finally by selecting a maximum spanning tree $\mathcal{T}$ on the cluster graph with edges between $\mathcal{C}_i$ and $\mathcal{C}_j$ if $\mathcal{C}_i \cap \mathcal{C}_j \neq \varnothing$ and edge weights equal to $|\mathcal{C}_i \cap \mathcal{C}_j|$. In the experiments, we used the *Min-Fill* greedy heuristic (it locally adds the minimum number of edges to the constraint graph), a very usual heuristic aimed at the production of tree-decompositions with a small tree-width [Rose, 1970].

In the following, from a tree-decomposition, we consider a rooted tree (I,F) with root $\mathcal{C}_1$ and we note $Sons(\mathcal{C}_i)$ the set of son clusters of $\mathcal{C}_i$ and $Desc(\mathcal{C}_i)$ the set of variables which belong to $\mathcal{C}_i$ or to any descendant $\mathcal{C}_j$ of $\mathcal{C}_i$ in the subtree rooted in $\mathcal{C}_i$.

Example 1. *Consider the CSP the constraint graph of which is provided in Fig.* 1, *a. We assume that each domain is* $\{a,b,c,d\}$ *and each constraint* $c_{ij} = \{x_i, x_j\}$ *has a relation* $r_{c_{ij}}$ *such that* $x_i \neq x_j$, *which defines a graph coloring problem.*



A – A constraint graph *b* – Its tree-decomposition
Fig. 1. A tree-decomposition of a small problem with 8 variables

Fig. 1, *b represents an optimal tree-decomposition for the chordal graph of Fig.* 1, *a. We have* $\mathcal{C}_1 = \{x_1, x_2, x_3\}$, $\mathcal{C}_2 = \{x_2, x_3, x_4, x_5\}$, $\mathcal{C}_3 = \{x_4, x_5, x_6\}$, *and* $\mathcal{C}_4 = \{x_3, x_7, x_8\}$. *For instance, Desc*($\mathcal{C}_2$)=$\mathcal{C}_2 \cup \mathcal{C}_2$= $= \{x_2, x_3, x_4, x_5, x_6\}$. *The tree-width is* 3.

3. Exact solution counting with #BTD

The essential property of a tree decomposition is that assigning $\mathcal{C}_i \cap \mathcal{C}_j$ ($\mathcal{C}_j \in Sons(\mathcal{C}_i)$) by the assignment $\mathcal{A}$ separates the initial problem into two subproblems, which can then be solved independently and the product of their number of solutions returned as the total number of solutions. The first subproblem, denoted $\mathcal{P}_{j/\mathcal{A}[\mathcal{C}_i \cap \mathcal{C}_j]}$ and rooted in $\mathcal{C}_j$, is defined by the variables in $Desc(\mathcal{C}_j)$, with variables $\mathcal{C}_i \cap \mathcal{C}_j$ assigned by $\mathcal{A}$, and by all the constraints involving *at least* one variable in $Desc(\mathcal{C}_j) \setminus \mathcal{C}_i$. The remaining constraints, together with the variables they involve, define the remaining subproblem.

A tree search backtracking algorithm can exploit this property by using a suitable variable assignment ordering: the variables of any cluster $\mathcal{C}_i$ must be assigned before the variables that remain in its son clusters. In this case, for any cluster $\mathcal{C}_j \in Sons(\mathcal{C}_i)$, once $\mathcal{C}_i \cap \mathcal{C}_j$ is assigned, the subproblem rooted in $\mathcal{C}_j$ conditioned by the current assignment $\mathcal{A}$ of $\mathcal{C}_i \cap \mathcal{C}_j$ can be solved independently of the rest of the problem. The exact number of solutions *nb* of this subproblem $\mathcal{P}_{j/\mathcal{A}[\mathcal{C}_i \cap \mathcal{C}_j]}$, called a *#good* and represented by a pair ($\mathcal{A}[\mathcal{C}_i \cap \mathcal{C}_j]$, *nb*), can be recorded, which means it will never be computed again for the same assignment of $\mathcal{C}_i \cap \mathcal{C}_j$. This is why algorithms such as BTD [Jégou and Terrioux, 2003] and AND/OR graph search [Dechter and Mateescu, 2004, 2007], exploiting the related notions of structural goods and *pseudo-tree* [Freuder and Quinn, 1985] respectively, are able to keep their time (and space) complexity exponential in the size of the largest cluster only.

We denote $\mathcal{S}_{j/\mathcal{A}}$ the number of solutions of subproblem $\mathcal{P}_{j/\mathcal{A}[\mathcal{C}_i \cap \mathcal{C}_j]}$ compatible with an assignment $\mathcal{A}$ of $(\mathcal{C}_i \cap \mathcal{C}_j) \cup Y$, $Y \subseteq \mathcal{C}_j$. It corresponds to the number of extensions of $\mathcal{A}$ on $Desc(\mathcal{C}_j)$ satisfying all the constraints in $\mathcal{P}_{j/\mathcal{A}[\mathcal{C}_i \cap \mathcal{C}_j]}$. The total number of solutions of $\mathcal{P}$ is $S_{1/\varnothing}$.

Example 2. *Consider the CSP in Example* 1. $(x_1, x_2, \ldots, x_8)$ *is a suitable variable ordering for the tree-decomposition of Fig.* 1, *b. Given* $\mathcal{A} = (x_1 \leftarrow a, x_2 \leftarrow b, x_3 \leftarrow c)$, *the variable set of* $\mathcal{P}_{2/\mathcal{A}[\mathcal{C}_1 \cap \mathcal{C}_2]}$ *is* $Desc(\mathcal{C}_2)$, (*with* $d_{x_2} = \{b\}, d_{x_3} = \{c\}$ *and* $d_{x_4} = d_{x_5} = d_{x_6} = \{a,b,c,d\}$) *and its constraint set is* $\{c_{24}, c_{25}, c_{34}, c_{35}, c_{45}, c_{46}, c_{56}\}$.

For instance, $\mathcal{S}_{2/\mathcal{A}} = \mathcal{S}_{3/(x_4 \leftarrow a, x_5 \leftarrow d)} + \mathcal{S}_{3/(x_4 \leftarrow d, x_5 \leftarrow a)} = 2 + 2 = 4$. *And the number of solutions of* $\mathcal{P}_{4/(x_3 \leftarrow c)}$ *is* $\mathcal{S}_{4/\mathcal{A}} = 6$. *Thus, there are* $\mathcal{S}_{2/\mathcal{A}} \times \mathcal{S}_{4/\mathcal{A}} = 24$ *extensions of* $\mathcal{A}$ *being solutions of* $\mathcal{P}$. *Note that for* $\mathcal{P}_{4/(x_3 \leftarrow c)}$, $((x_3 \leftarrow c), 6)$ *is a #good. So, for any other assignment* $\mathcal{A}'$ *of* $\mathcal{C}_1$ *with* x_3 *assigned to c, it is not necessary to compute the number of solutions of* $\mathcal{P}_{4/\mathcal{A}'[\mathcal{C}_1 \cap \mathcal{C}_4]}$ *because the #good* $((x_3 \leftarrow c), 6)$ *will be exploited in this case. The total number of solutions is* $\mathcal{S}_{1/\varnothing} = 576$.

#BTD is described in Algorithm 1. Given an assignment $\mathcal{A}$, a cluster $\mathcal{C}_i$, and a set $V_{\mathcal{C}_i}$ of unassigned variables of $\mathcal{C}_i$, #BTD ($\mathcal{A}, \mathcal{C}_i, V_{\mathcal{C}_i}$) looks for the number $\mathcal{S}_{i/\mathcal{A}}$ of extensions $\mathcal{B}$ of $\mathcal{A}$ on $Desc(\mathcal{C}_i)$ such that $\mathcal{A}[\mathcal{C}_i \setminus V_{\mathcal{C}_i}] = \mathcal{B}[\mathcal{C}_i \setminus V_{\mathcal{C}_i}]$. The first call is #BTD ($\varnothing, \mathcal{C}_1, \mathcal{C}_1$) and it returns the number of solutions $\mathcal{S}_{1/\varnothing}$. Inside a cluster $\mathcal{C}_i$, it proceeds classically by assigning a value to a variable and by backtracking if any constraint is violated. When every variable in $\mathcal{C}_i$ is assigned, #BTD computes the number of solutions of the subproblem rooted in the first son of $\mathcal{C}_i$, if there is one (otherwise the current subproblem is totally assigned and contains only one solution). More generally, let us consider $\mathcal{C}_j$, a son of $\mathcal{C}_i$. Given a current assignment $\mathcal{A}$ of $\mathcal{C}_i$, #BTD checks whether the assignment $\mathcal{A}[\mathcal{C}_i \cap \mathcal{C}_j]$ corresponds to a #good. If so, #BTD multiplies the recorded number of solutions with the current number of solutions *NbSol* ($\mathcal{S}_{i/\mathcal{A}}$). Otherwise, it extends $\mathcal{A}$ on $Desc(\mathcal{C}_j)$ in order to compute its number of consistent extensions nb ($\mathcal{S}_{j/\mathcal{A}}$). Then, it records the #good $(\mathcal{A}[\mathcal{C}_i \cap \mathcal{C}_j], nb)$. #BTD computes the number of solutions of the subproblem rooted in the next son of $\mathcal{C}_i$. Finally, when each son of $\mathcal{C}_i$ has been examined, #BTD tries to modify the current assignment of $\mathcal{C}_i$. The number of solutions of the subproblem rooted in $\mathcal{C}_i$ is the sum of solution counts for every assignment of $\mathcal{C}_i$.

Theorem 1. *#BTD is sound, complete and terminates.*

Proof. #BTD exploits two kinds of problem decomposition. The first one is based on conditioning. The second one is based on tree-decomposition. In the first case (Else branch starting at line 2 in Algorithm 1), let $\mathcal{P}^x_a$ be the subproblem derived from $\mathcal{P}$ by assigning variable x to value a. We have $\mathcal{P} = \bigcup_{a \in d_x} \mathcal{P}^x_a$. We denote $Sol_{\mathcal{P}}$ the set of solutions of $\mathcal{P}$ and $\mathcal{S}_{\mathcal{P}} = |Sol_{\mathcal{P}}|$. For any two distinct values a and b of d_x, we have $Sol_{\mathcal{P}^x_a} \cap Sol_{\mathcal{P}^x_b} = \varnothing$. Thus, the set $\{Sol_{\mathcal{P}^x_a} \mid a \in d_x\}$ is a partition of $Sol_{\mathcal{P}}$ and $\mathcal{S}_{\mathcal{P}} = \sum_{a \in d_x} \mathcal{S}_{\mathcal{P}^x_a}$.

In the second case of problem decomposition (If branch starting at line 1 in Algorithm 1), we are dealing with independent subproblems. Two CSPs $\mathcal{P}_1 = (X_1, D_1, C_1, R_1)$ and $\mathcal{P}_2 = (X_2, D_2, C_2, R_2)$ are independent if and only if $X_1 \cap X_2 = \varnothing$. If $\mathcal{P} = \mathcal{P}_1 \cup \mathcal{P}_2$ with $\mathcal{P}_1$ and $\mathcal{P}_2$ two independent subproblems, then the solutions of $\mathcal{P}$ is the Cartesian product of the solutions of $\mathcal{P}_1$ and $\mathcal{P}_2$. Therefore, $\mathcal{S}_{\mathcal{P}} = \mathcal{S}_{\mathcal{P}_1} \times \mathcal{S}_{\mathcal{P}_2}$. In the case of a tree-decomposition, given a cluster $\mathcal{C}_i$ and an assignment $\mathcal{A}$ of $Y \subset X$ such that $Desc(\mathcal{C}_i) \cap Y = \mathcal{C}_i$, we have all the subproblems $\mathcal{P}_{j/\mathcal{A}[\mathcal{C}_i \cap \mathcal{C}_j]}, \forall \mathcal{C}_j \in$ $\in Sons(\mathcal{C}_i)$ mutually independent. Then $\mathcal{A}$ has $\mathcal{S}_{i/\mathcal{A}} = \prod_{\mathcal{C}_j \in Sons(\mathcal{C}_i)} \mathcal{S}_{j/\mathcal{A}}$ consistent extensions on $Desc(\mathcal{C}_i)$.

If (I, nb) is a #good of $\mathcal{P}_{j/\mathcal{A}[\mathcal{C}_i \cap \mathcal{C}_j]}$ such that $\mathcal{A}[C_i \cap C_j] = \mathcal{T}$, then $\mathcal{A}$ has nb consistent extensions on $Desc(C_j)$: $\mathcal{S}_{j/\mathcal{A}} = nb$. □

Algorithm 1: #BTD($\mathcal{A}$, $\mathcal{C}_i$, $V_{\mathcal{C}_i}$): integer

```
    if $V_{\mathcal{C}_i} = \varnothing$ then
1       if $Sons(\mathcal{C}_i) = \varnothing$ then return 1;
        else
            $S \leftarrow Sons(\mathcal{C}_i)$;
            $NbSol \leftarrow 1$
            while $S \neq \varnothing$ and $NbSol \neq 0$ do
                choose $\mathcal{C}_j$ in $S$;
                $S \leftarrow S \setminus \{\mathcal{C}_j\}$;
                if $(\mathcal{A}[\mathcal{C}_i \cap \mathcal{C}_j], nb)$ is a #good of $\mathcal{P}_{j/\mathcal{A}[\mathcal{C}_i \cap \mathcal{C}_j]}$ then
                    $NbSol \leftarrow NbSol \times nb$;
                else
                    $nb \leftarrow$ #BTD($\mathcal{A}, \mathcal{C}_j, \mathcal{C}_j \setminus (\mathcal{C}_i \cap \mathcal{C}_j)$);
                    record #good $(\mathcal{A}[\mathcal{C}_i \cap \mathcal{C}_j], nb)$ of $\mathcal{P}_{j/\mathcal{A}[\mathcal{C}_i \cap \mathcal{C}_j]}$;
                    $NbSol \leftarrow NbSol \times nb$;

            return $NbSol$;
    else
2       choose $x \in V_{\mathcal{C}_i}$;
        $d \leftarrow d_x$;
        $NbSol \leftarrow 0$;
        while $d \neq \varnothing$ do
            choose $a$ in $d$;
            $d \leftarrow d \setminus \{a\}$;
3           if $\mathcal{A} \cup (x \leftarrow a)$ does not violate any $c \in C$ then
                $NbSol \leftarrow NbSol +$ #BTD($\mathcal{A} \cup (x \leftarrow a)$,
                                        $\mathcal{C}_i$, $V_{\mathcal{C}_i} \setminus \{x\}$);
        return $NbSol$;
```

Theorem 2. *#BTD has time complexity in $\mathcal{O}(n \cdot m \cdot d^{w+1})$ and space complexity in $\mathcal{O}(n \cdot s \cdot d^s)$.*

Proof. *Space complexity.* #BTD only records #goods. These are assignments on the intersections $\mathcal{C}_i \cap \mathcal{C}_j$ with $\mathcal{C}_j$ a son of $\mathcal{C}_i$. Therefore, if s is the size of the largest of these intersections, #BTD has a space complexity of $\mathcal{O}(n \cdot s \cdot d^s)$ because the number of these intersections is bounded by n, while the number of #goods associated to one intersection is bounded by d^s and the size of a #good is at most s.

Time complexity. In the worst case, #BTD explores all the clusters (at most n) and tries all the values of every variable inside each cluster, each time checking at most m constraints at line 3.

Thanks to its #good recording mechanism, it never explores the same cluster with the same assignment of its variables twice. The number of assignments of a cluster is bounded by d^{w+1} with $w = \max_{\mathcal{C}_i \in \mathcal{C}} |\mathcal{C}_i| - 1$, the width of the tree-decomposition. Consequently, #BTD has a time complexity in $\mathcal{O}(n \cdot m \cdot d^{w+1})$.

In practice, for problems with a large tree-width, #BTD may run out of time and memory, as shown in Section 5. In this case, we are interested in an approximate method.

4. Approximate solution counting with Approx#BTD

We consider here CSPs with a large tree-width but a sparse constraint graph. We define a collection of *easy-to-solve* subproblems of an original problem $\mathcal{P}$ by partitioning the set of constraints, that is the set of edges in the constraint graph in the case of a binary CSP. The constraint graph (X,C) will be partitioned into k subgraphs (X_1,E_1), … , (X_k,E_k), such that $\cup X_i = X$, $\cup E_i = C$ and $\cap E_i = \varnothing$. We add the extra property that each (X_i,E_i) is chordal (without adding extra edges as for building a tree-decomposition). Thus, each (X_i,E_i) will be associated to a chordal subproblem $\mathcal{P}_i$ (with corresponding sets of variables X_i and constraints E_i), which should have a small tree-width and be efficiently solved using #BTD.

Assume that $\mathcal{S}_{\mathcal{P}_i}$ is the number of solutions for each subproblem $\mathcal{P}_i$, $1 \leq i \leq k$. We will estimate the number of solutions of $\mathcal{P}$ exploiting the following property. Let $\mathcal{A}$ be any assignment of *X*, we denote $\mathbb{P}(\mathcal{A} \vDash \mathcal{P})$ the probability of «*$\mathcal{A}$ is a solution of $\mathcal{P}$*». We have $\mathbb{P}(\mathcal{A} \vDash \mathcal{P}) = \frac{\mathcal{S}_{\mathcal{P}}}{\prod_{x \in X} d_x}$, assuming a uniform prior probability distribution among the different value assignments. We also have

$$\mathbb{P}(\mathcal{A} \vDash \mathcal{P}) = \mathbb{P}(\mathcal{A} \vDash \mathcal{P}_1 \wedge \mathcal{A} \vDash \mathcal{P}_2 \wedge \ldots \wedge \mathcal{A} \vDash \mathcal{P}_k) =$$

$$= \mathbb{P}(\mathcal{A} \vDash \mathcal{P}_1)\mathbb{P}(\mathcal{A} \vDash \mathcal{P}_2 \mid \mathcal{A} \vDash \mathcal{P}_1)\ldots$$
$$\ldots\mathbb{P}(\mathcal{A} \vDash \mathcal{P}_k \mid \mathcal{A} \vDash \mathcal{P}_1 \wedge \ldots \wedge \mathcal{A} \vDash \mathcal{P}_{k-1}).$$

In order to simplify these conditional probabilities, we assume probability independence between the $(\mathcal{A} \vDash \mathcal{P}_i)$ terms, which is true only if $\cap X_i = \varnothing$. Thus, we have

$$\mathbb{P}(\mathcal{A} \vDash \mathcal{P}) \approx \mathbb{P}(\mathcal{A} \vDash \mathcal{P}_1)\mathbb{P}(\mathcal{A} \vDash \mathcal{P}_2)\ldots\mathbb{P}(\mathcal{A} \vDash \mathcal{P}_k).$$

Now, we can easily deduce the following property in order to estimate $\mathcal{S}_{\mathcal{P}}$.

Property 1. *Given a CSP $\mathcal{P} = (X,D,C,R)$ and a partition $\{\mathcal{P}_1,...,\mathcal{P}_k\}$ of $\mathcal{P}$ induced by a partition of C in k elements.*

$$\mathcal{S}_{\mathcal{P}} \approx \left\lceil \left(\prod_{i=1}^{k} \frac{\mathcal{S}_{\mathcal{P}_i}}{\prod_{x \in X_i} d_x} \right) \times \prod_{x \in X} d_x \right\rceil. \qquad (1)$$

Recall that this approximation returns an exact answer if all the subproblems are independent ($\cap X_i = \varnothing$) or $k = 1$ ($\mathcal{P}$ is already chordal as in Example 1) or if there exists an inconsistent subproblem $\mathcal{P}_i$ ($\mathcal{P}$ has no solution)[3]. Moreover, we can provide a trivial upper bound on the number of solutions due to the fact that each subproblem $\mathcal{P}_i$ is a relaxation of $\mathcal{P}$ (the same argument is used in [Pesant, 2005] to construct an upper bound):

$$\mathcal{S}_{\mathcal{P}} \leq \min_{i \in [1,k]} \frac{\mathcal{S}_{\mathcal{P}_i}}{\prod_{x \in X_i} d_x} \times \prod_{x \in X} d_x \qquad (2)$$

`Approx#BTD` is described in Algorithm 2. Applied to a problem $\mathcal{P}$ with constraint graph (X,C), the method builds a partition $\{E_1,...,E_k\}$ of C such that the constraint graph (X_i,E_i) is chordal for all $1 \leq i \leq k$. Each chordal subgraph is produced by the `MaxChord`$^+$ algorithm[4] [Dear-

[3]Due to the celling function in Equation 1, if the approximation returns zero then $\mathcal{P}$ has no solution.

[4]`MaxChord`$^+$ returns a maximal subgraph for binary CSPs. For non-binary CSPs, we do not guarantee subgraph maximality and add to the subproblem all the constraints totally included in the extracted chordal subgraph. In Figure 2, the edge $\{x_3,x_5\}$ has been removed from the first part be-

ing et al., 1988], described in Algorithm 3. An example of a partition found by Approx#BTD is given in Figure 2. Subproblems associated to (X_i, E_i) are solved with #BTD. The method returns an approximation to the number of solutions of $\mathcal{P}$ based on Property 1.

Algorithm 2: Approx#BTD(X,C): integer

```
$G' \leftarrow (X,C)$;
$i \leftarrow 0$;
while $G' \neq (\varnothing,\varnothing)$ do
    $i \leftarrow i+1$;
    $(X',C') \leftarrow G'$;
    $(X_i,E_i) \leftarrow$ MaxChord$^+(X',C')$;
    Let $\mathcal{P}_i$ be the subproblem associated with $(X_i,E_i)$;
    $\mathcal{S}_{\mathcal{P}_i} \leftarrow$ #BTD$(\varnothing,\mathcal{C}_1^i,\mathcal{C}_1^i)$ with $\mathcal{C}_1^i$ the root cluster
        of $\mathcal{P}_i$ optimal tree-decomposition;
    $G' \leftarrow (X'',C' \setminus E_i)$ with $X''$ be the set of variables
        induced by $C' \setminus E_i$;
$k \leftarrow i$;
return $\left\lceil \left( \prod_{i=1}^{k} \frac{\mathrm{S}_{\mathrm{P}_i}}{\prod_{x\in X_i} d_x} \right) \times \prod_{x\in X} d_x \right\rceil$;
```

Theorem 3. *Approx#BTD is sound, complete and terminates.*

Proof. It suffices to prove that we have a partition of the constraints at the end of the **while** loop in order to be able to apply Property 1. This can be easily shown by induction using the invariants $X = X' \cup \left(\cup_{j=1}^{i} X_j\right)$ and «$Q = (C', E_1, E_2, \dots, E_i)$ is a partition of C» inside the loop.

Theorem 4. *Approx#BTD has time complexity in $\mathcal{O}(n^2 \cdot m \cdot d^{w'+1})$ and space complexity in $\mathcal{O}(n \cdot s' \cdot d^{s'})$ with $s' < w'+1 \leq c \leq w+1 \leq n$.*

Proof. *Space complexity.* Approx#BTD has the same space complexity as #BTD applied on the largest subproblem $\mathcal{P}_i$ w.r.t. the largest cluster intersection denoted $s' = \max_{\mathcal{C}_u^i,\mathcal{C}_v^i \in \mathcal{C}^i, u\neq v, i\in[1,k]} |\mathcal{C}_u^i \cap \mathcal{C}_v^i|$.

Time complexity. The number of iterations of Approx#BTD is less than *n*. At each step, the first variable considered by MaxChord$^+$ at line 4 will have all its constraints totally included in the maximal chordal subgraph. Each chordal subgraph and its associated optimal tree-decomposition can be computed in $O(nm)$ [Dearing et al., 1988]. Thus, the time complexity of Approx#BTD is in $\mathcal{O}(n^2 \cdot m \cdot d^{w'+1})$ with the largest subproblem tree-width $w' = \max_{\mathcal{C}_u^i \in \mathcal{C}^i, i\in[1,k]} |\mathcal{C}_u^i| - 1$. Because each $\mathcal{P}_i$ is a partial chordal subgraph of $\mathcal{P}$, its tree-width w' is equal to the maximum clique size in its subgraph [Fulkerson and Gross, 1965] which is by definition less than or equal to the maximum clique size of the original problem, called the clique number c, inferior to the problem tree-width $w+1$. □

Algorithm 3: MaxChord$^+(X,C)$:
(set of variables, set of constraints)

```
    /* Original MaxChord algorithm
    foreach $v \in X$ do $Y(v) \leftarrow \varnothing$;
4   Choose $v_0 \in X$;
    $S \leftarrow \{v_0\}$;
    $C' \leftarrow \varnothing$;
    $l \leftarrow |X|$;
    while $l > 1$ do
        forall $u \in X \setminus S$ such as $\{u,v_0\} \subseteq c \in C$ do
            if $Y(u) \subseteq Y(v_0)$ then
                $Y(u) \leftarrow Y(u) \cup \{v_0\}$;
                $C' \leftarrow C' \cup \{u,v_0\}$;
        Choose $v_0 = argmax_{v\in X\setminus S} |Y(v)|$;
        $S \leftarrow S \cup \{v_0\}$;
        $l \leftarrow l-1$;
    /*Additional part: selection of all the (non-binary) constraints
        of $C$ embedded in the maximal chordal subgraph $(X,C')$
    $C'' \leftarrow \varnothing$;
    foreach $c \in C$ do
        if $\forall\{u,v\} \subseteq c, \{u,v\} \in C'$ do;
            $C'' \leftarrow C'' \cup \{c\}$;
    return $(X,C'')$;
```

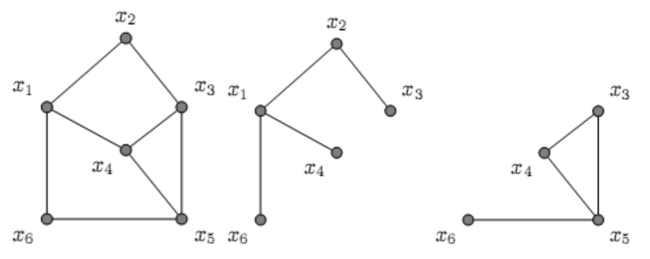
5. The experimental results

We implemented #BTD and Approx#BTD counting methods on top of toulbar2 C++ solver[5]. The experimentations were performed on

cause it is associated to the ternary constraint {x_3, x_4x_5} not totally included in this part.

[5]http://mulcyber.toulouse.inra.fr/projects/toulbar2/ version 0.8.1.

a Linux 2.6 GHz Intel Xeon computer with 2GB. Reported times (total CPU times as given by the #SAT solvers or reported by the bash command «time» if not) are in seconds. For `#BTD` and `Approx#BTD`, the total time does not include the task of finding a variable elimination ordering[6]. We limit to one hour the time spent for solving a given instance. Inside `#BTD` (line 3), we use generalized arc consistency (only for constraints with 2 or 3 unassigned variables) instead of backward checking, for efficiency reasons. The *min domain /max degree* dynamic variable ordering, modified by a conflict back-jumping heuristic [Lecoutre et al., 2006], is used inside the clusters. Our methods exploit a binary branching scheme. The variable is assigned to its first value or this value is removed from the domain.



a – An original problem *b* – The first part *c* – The second part

Fig. 2. A partition of a CSP with 6 variables found by `Approx#BTD`. The original problem has 5 binary constraints ($\{x_1,x_2\}$, $\{x_1,x_4\}$, $\{x_1,x_6\}$, $\{x_2,x_3\}$, $\{x_5,x_6\}$), and one ternary constraint $\{x_3,x_4,x_5\}$. We have $k = 2$, $w' = 2$, $c = w = 3$

We performed experiments on SAT and CSP benchmarks[7]. We selected academic (random k-SAT *wff*, All-Interval Series *ais*, Towers of Hanoi *hanoi*) and industrial (circuit fault analysis *ssa* and *bit*, logistics planning *logistics*) satisfiable instances. CSP benchmarks are graph coloring instances (counting the number of optimal solutions). For #SAT solvers only, CSP instances are translated into SAT by using the direct encoding (one Boolean variable per domain value, one clause per domain to enforce at least one domain value is selected, and a set of binary clauses to forbid multiple value selection).

[6]Which was always fast to compute (linear complexity).

[7]From www.satlib.org, www.satcompetition.org and mat.gsia.cmu.edu/COLOR02/.

5.1. The evaluation of exact methods

We compared `#BTD` with state-of-the-art #SAT solvers Relsat [Bayardo and Pehoushek, 2000] v2.02, `Cachet` [Sang et al., 2004] v1.22 (with a default memory limit of 5 MB), `sharpSAT` [Thurley, 2006] v1.1, and also `c2d` [Darwiche, 2004] v2.20 which also exploits the problem structure (with a default memory limit of 512 MB and a limit of 64 MB for storing its d-DNNF). `c2d` and `#BTD` methods use the *Min-Fill* variable elimination ordering heuristic (except for *hanoi* where we used the default file order) to construct a tree-decomposition / d-DNNF.

Our results are summarized in Table 1. The co

T a b l e 1. Comparison of exact methods. Legend: mem: out of memory, – : out of time (for `c2d`, * : out of memory for storing NNF)

$\mathcal{P}$	*n* (Bool-vars)	*d*	*w*	$\mathcal{S}_\mathcal{P}$	`c2d`	`sharpSAT`	`Cachet`	`Relsat`	`#BTD`
					Time	Time	Time	Time	Time
SAT									
wff.3.100.150	100		39	1.8e21	*	mem	–	–	mem
wff.3.150.525	150		92	1.4e14	*	mem	**266.**	2509	mem
wff.4.100.500	100		80	–	*	mem	–	–	mem
ssa7552-038	1501		25	2.84e40	0.15	**0.06**	0.22	67	0.65
ssa7552-158	1363		9	2.56e31	0.10	**0.03**	0.07	3	0.19
ssa7552-159	1363		11	7.66e33	0.09	**0.04**	0.07	4	0.27
ssa7552-160	1391		12	7.47e32	0.12	**0.04**	0.08	5	0.30
2bitcomp_5	125		36	9.84e15	0.43	**0.05**	0.14	1	16.24
2bitmax_6	252	2	58	2.10e29	17.00	**0.87**	1.51	20	mem
ais6	61		41	24	0.05	**0.01**	0.03	<1	0.08
ais8	113		77	40	0.51	**0.17**	0.58	<1	3.27
ais10	181		116	296	16.64	**4.13**	29.19	6	543
ais12	265		181	1328	1147	**161.**	2173	229	–
logistics.a	828		116	3.8e14	–	**0.17**	3.78	10	mem
logistics.b	843		107	2.3e23	–	**1.38**	12.34	433	mem
hanoi4	718		46	1	7.18	**1.11**	32.64	3	1.87
hanoi5	1931		58	1	–	mem	–	–	**26.75**
CSP (Graph Coloring)									
2-Insertions_3	37 (148)	4	9	6.84e13	235.	mem	–	–	**7.80**
2-Insertions_4	149 (596)	4	38	–	–	mem	–	–	–
DSJC125.1	125 (625)	5	65	–	–	mem	–	–	mem
games120	120 (1080)	9	41	–	–	mem	–	–	mem
GEOM30a	30 (180)	6	6	4.98e14	0.86	5.53	–	–	**0.10**
GEOM40	40 (240)	6	5	4.1e23	1.00	mem	–	–	**0.09**
le450_5a	450 (2250)	5	315	3840	–	**32.31**	318	326	1100
le450_5b	450 (2250)	5	318	120	–	**13.12**	227	187	1364
le450_5c	450 (2250)	5	315	120	–	**2.18**	19.09	57	47.53
le450_5d	450 (2250)	5	299	960	–	**4.40**	14.60	36	92.03
Mug100_1	100 (400)	4	3	1.3e37	0.19	23.88	–	–	**0.02**
myciel5	47 (282)	6	21	–	–	mem	–	–	mem

lumns are: instance name, number of variables (and also the number of Boolean variables on translated

CSP instances), maximum domain size, width of the tree-decomposition, exact number of solutions if known, time for `c2d`, `sharpSAT`, `Cachet`, `Relsat`, and `#BTD`. We noticed that `#BTD` can solve instances with relatively small tree-widths (except for *ais* and *le450* which have few solutions). Exact #SAT solvers generally perform better than `#BTD` on SAT instances (except for *hanoi5*), with `sharpSAT` obtaining the best results, but have difficulties on translated CSP instances. Here, `#BTD` maintaining arc consistency performed better than #SAT solvers using unit propagation.

5.2. Evaluation of approximate methods

Table 2 gives an analysis of `Approx#BTD` on the tested instances. The columns are: instance name, number of variables, maximum domain size, exact number of solutions if known, width of the tree-decomposition for the original problem, maximum width of the tree-decomposition for all the chordal subproblems, number of subproblems in the partition, approximate number of solutions and upper-bound on the number of solutions as given by Equation 2, and time for `Approx#BTD`.

T a b l e 2. Analysis of `Approx#BTD` performance and subproblem features

$\mathcal{P}$	n	d	$\mathcal{S}_\mathcal{P}$	w	w'	k	$\mathcal{S}_\mathcal{P}$		Time
SAT									
wff.3.100.150	100		1.80e21	39	3	6	≈ 3.10e21	≤ 5.05e27	0.03
wff.3.150.525	150		1.14e14	92	3	13	≈ 1.58e15	≤ 1.13e42	0.18
wff.4.100.500	100		–	80	6	48	≈ 1.59e16	≤ 4.87e29	0.47
ssa7552-038	1501		2.84e40	25	6	4	≈ 9.33e38	≤ 3.79e142	1.34
ssa7552-158	1363		2.56e31	9	5	3	≈ 2.22e25	≤ 1.41e79	0.77
ssa7552-159	1363		7.66e33	11	5	3	≈ 6.53e27	≤ 6.33e88	0.85
ssa7552-160	1391		7.47e32	12	5	3	≈ 4.50e26	≤ 3.36e106	1.09
2bitcomp_5	125		9.84e15	36	6	4	≈ 8.61e16	≤ 6.81e27	0.02
2bitmax_6	252	2	2.10e29	58	6	4	≈ 4.53e29	≤ 7.19e45	0.10
ais6	61		24	41	12	8	≈ 1	≤ 1.81e9	0.04
ais8	113		40	77	16	11	≈ 1	≤ 3.49e15	0.20
ais10	181		296	116	21	23	≈ 1	≤ 2.06e22	0.84
ais12	265		1328	181	26	23	≈ 1	≤ 3.19e29	2.47
logistics.a	828		3.8e14	116	10	24	≈ 1	≤ 4.91e180	14.85
logistics.b	843		2.3e23	107	13	25	≈ 1	≤ 2.15e169	14.08
hanoi4	718		1	46	10	8	≈ 1	≤ 4.26e106	1.40
hanoi5	1931		1	58	12	11	≈ 1	≤ 4.48e309	16.05
CSP (Graph Coloring)									
2-Insertions_3	37	4	6.84e13	9	1	3	≈ 1.91e13	≤ 6.00e17	0.01
2-Insertions_4	149	4	–	38	1	6	≈ 1.30e22	≤ 1.64e71	0.07
DSJC125.1	125	5	–	65	3	7	≈ 1.23e13	≤ 2.27e70	0.12
games120	120	9	–	41	8	6	≈ 1.12e78	≤ 1.92e99	9.84
GEOM30a	30	6	4.98e14	6	5	2	≈ 7.29e14	≤ 1.81e15	0.04
GEOM40	40	6	4.1e23	5	5	2	≈ 4.8e23	≤ 1.72e24	0.01
le450_5a	450	5	3840	315	4	13	≈ 1	≤ 2.41e216	3.17
le450_5b	450	5	120	318	4	13	≈ 1	≤ 5.72e213	3.23
le450_5c	450	5	120	315	4	20	≈ 1	≤ 1.49e201	7.42
le450_5d	450	5	960	299	4	20	≈ 1	≤ 8.58e200	7.38
mug100_1	100	4	1.3e37	3	2	2	≈ 5.33e37	≤ 7.18e41	0.01
myciel5	47	6	–	21	1	8	≈ 7.70e17	≤ 8.53e32	0.03
myciel6	95	7	–	35	1	13	≈ 5.49e29	≤ 9.80e73	0.19
myciel7	191	8	–	66	1	21	≈ 4.25e35	≤ 2.96e161	1.23

Our approximate method exploits a partition of the constraint graph in such a way that the resulting subproblems to solve have a small tree-width on these instances ($w' \le 26$). It has the practical effect that the method is relatively fast whatever the original tree-width. Notice that the upper-bound is generally very poor even with a small number of subproblems (*e.g. ssa*).

We also compared `Approx#BTD` with the approximation method `SampleCount` [Gomes et al., 2007a]. With parameters ($s = 20$, $t = 7$, $\alpha = 1$), `SampleCount-LB` provides an estimated lower bound on the number of solutions with a high-confidence interval (99% confidence), after seven runs. With parameters ($s = 20$, $t = 1$, $\alpha = 0$), called `SampleCount-A` in the following table, it gives only an approximation without any guarantee, after the first run of `SampleCount-LB`.

Our results are summarized in Table 3. The columns are : instance name, exact number of solutions if known, approximate number of solutions and time for `Approx#BTD` and `SampleCount-A`, and estimated lower bound on the number of solutions and time for `SampleCount-LB`. The qua-

T a b l e 3. Comparison of approximate methods. Legend: – : out of time (1 hour)

$\mathcal{P}$	$\mathcal{S}_\mathcal{P}$	Approx#BTD		Sample Count-A		Sample Count-LB	
		$\hat{\mathcal{S}}_\mathcal{P}$	Time	$\hat{\mathcal{S}}_\mathcal{P}$	Time	$\hat{\mathcal{S}}_\mathcal{P}$	Time
SAT							
wff.3.100.150	1.8e21	≈ 3.10e21	0.1	≈ **1.37e21**	959.8	–	–
wff.3.150.525	1.4e14	≈ 1.58e15	0.2	≈ **3.80e14**	0.7	≥ 2.53e12	4.6
wff.4.100.500	–	≈ 1.59e16	0.5	≈ 4.15e16	2045.0	–	–
ssa7552-038	2.84e40	≈ 9.33e38	1.3	≈ **1.11e40**	134.0	≥ 3.54e38	1162.0
ssa7552-158	2.56e31	≈ 2.22e25	0.8	≈ **1.43e30**	14.1	≥ **1.43e30**	177.0
ssa7552-159	7.66e33	≈ 6.53e27	0.8	≈ 6.49e34	32.8	≥ **1.64e32**	182.0
ssa7552-160	7.47e32	≈ 4.50e26	1.1	≈ **5.08e32**	144.0	≥ 2.31e31	1293.0
2bitcomp_5	9.84e15	≈ 8.61e16	0.1	≈ **4.37e15**	0.2	≥ 3.26e15	1.2
2bitmax_6	2.10e29	≈ 4.53e29	0.1	≈ **1.62e29**	1.7	≥ 2.36e26	10.3
ais6	24	≈ 1	0.1	≈ **12**	0.1	≥ **12**	0.2
ais8	40	≈ 1	0.2	≈ **16**	1.5	≥ 12	15.9
ais10	296	≈ 1	0.8	≈ **124**	45.9	≥ 20	312.0
ais12	1328	≈ **1**	2.5	≈ 0	9.2	≥ 0	9.2
logistics.a	3.8e14	≈ 1	14.8	≈ **7.25e11**	171.0	≥ 0	605.0
logistics.b	2.3e23	≈ 1	14.1	≈ **2.13e23**	199.0	≥ 0	229.0
hanoi4	1	≈ **1**	1.4	≈ 0	5.2	≥ 0	5.2
hanoi5	1	≈ **1**	16.0	≈ 0	6.1	≥ 0	6.2

$\mathcal{P}$	$\mathcal{S}_{\mathcal{P}}$	Approx#BTD		SampleCount-A		SampleCount-LB	
		$\hat{\mathcal{S}}_{\mathcal{P}}$	Time	$\hat{\mathcal{S}}_{\mathcal{P}}$	Time	$\hat{\mathcal{S}}_{\mathcal{P}}$	Time
CSP (Graph Coloring)							
2-Insertions_3	6.84e13	≈ **1.91e13**	0.1	≈ 4.73e12	1.0	≥ 4.73e12	7.4
2-Insertions_4	–	≈ 1.30e22	0.1	≈ 0	3.8	≥ 0	3.8
DSJC125.1	–	≈ 1.23e13	0.1	≈ 0	73.1	≥ 0	73.2
games120	–	≈ 1.12e78	9.8	≈ 7.33e64	13.8	≥ 1.35e61	91.1
GEOM30a	4.98e14	≈ **7.29e14**	0.1	≈ 1.23e13	0.4	≥ 3.28e12	3.7
GEOM40	4.1e23	≈ **4.8e23**	0.1	≈ 2.14e20	1.5	≥ 6.50e19	9.3
le450_5a	3840	≈ **1**	3.2	≈ 0	8.6	≥ 0	8.6
le450_5b	120	≈ **1**	3.2	≈ 0	8.6	≥ 0	8.6
le450_5c	120	≈ **1**	7.4	≈ 0	110.0	≥ 0	111.0
le450_5d	960	≈ **1**	7.4	≈ 0	4.6	≥ 0	54.6
mug100_1	1.3e37	≈ **5.33e37**	0.1	≈ 4.2e34	2.1	≥ 4.20e34	15.6
myciel5	–	≈ 7.69e17	0.1	≈ 7.29e17	0.9	≥ 7.29e17	6.4
myciel6	–	≈ 5.49e29	0.2	≈ 9.38e40	4.5	≥ 7.42e36	30.7
myciel7	–	≈ 4.26e35	1.2	≈ 1.37e80	27.7	≥ 5.56e74	163.0

lity of the approximation found by `Approx#BTD` is relatively good and it is comparable (except for *ssa*, *logistics*, and *myciel*6-7 benchmarks) to `SampleCount-A`, which takes more time.

For graph coloring, `Approx#BTD` outperforms also a dedicated CSP approach producing an estimated lower bound: 2_*Insertion*_3 ≥ 2.3e12, *games1*20 ≥ 4.5e42, and *mug*100_1 ≥ 1.0e28 in 1 minute each; *myciel*5 ≥ 4.1e17 in 12 minutes, times were measured on a 3.8GHz Xeon as reported in [Gomes et al., 2007b].

6. Conclusion

In this paper, we have proposed two methods for counting solutions of CSPs. These methods are based on a structural decomposition of CSPs. We have presented an exact method, which is adapted to problems with a small tree-width. For problems with a large tree-width and a sparse constraint graph, we have presented a new approximate method whose quality is comparable with existing methods, which is much faster than other approaches, and which requires no parameter tuning (except for the choice of a tree decomposition heuristic). Exploring other structural parameters [Nishimura et al., 2007, Samer and Szeider, 2010] should deserve future work. A practical improvement of our approach would be to impose a limit on the maximum clique size of the extracted chordal subproblems when the original problem has large arity constraints or a large clique number. Conversely, denser non-chordal subproblems could be produced and solved in an anytime manner as done in [Choi and Darwiche,2006] when the original problem has a small clique number.

A direction of future work is also to extend our approach to the problem of (approximate) inference in probabilistic discrete graphical models.

1. *Arnborg S.*, *Corneil D.*, *Proskurowski A.* Complexity of finding embeddings in a k-tree // SIAM J. of Discrete Mathematics. – 1987. – N 8. – P. 277–284.
2. *Bayardo R.*, *Pehoushek J.* Counting models using connected components // Proc. of AAAI-00. – Austin, Texas. – 2000. – P. 157–162.
3. *Burton R.*, *Steif J.* Nonuniqueness of measures of maximal entropy for subshifts of finite type // Ergodic theory and dynamical system, 1994.
4. *Choi A.*, *Darwiche A.* An edge deletion semantics for belief propagation and its practical impact on approximation quality // Proc. of AAAI-06. – 2006. – P. 1107–1114.
5. *Darwiche A.* On the tractable counting of theory models and its applications to truth maintenance and belief revision // J. of Applied Non-classical Logic. – 2001. – N 11. – P. 11–34.
6. *Darwiche A.* New advances in compiling cnf to decomposable negation normal form // Proc. of ECAI-04. Valencia, Spain. – 2004. – P. 328–332.
7. *Dearing P.*, *Shier D.*, *Warner D.* Maximal chordal subgraphs // Discrete Applied Mathematics. – 1988. – **20**(3). – P. 181–190.
8. *Dechter R.*, *Mateescu R.* And/or search spaces for graphical models // Artif. Intell. – 2007. – **171**(2–3). – P. 73–106.
9. *Dechter R.*, *Mateescu R.* The impact of and/or search spaces on constraint satisfaction and counting // Proc. of CP-04. Toronto, CA. – 2004. – P. 731–736.
10. *Favier A.*, *de Givry S.*, *Jégou P.* Exploiting problem structure for solution counting // Proc. of CP-09, Lisbon, Portugal. – 2009. – P. 335–343.
11. *Freuder E.*, *Quinn M.* Taking advantage of stable sets of variables in constraint satisfaction problems // Proc. of IJCAI-85. Los Angeles, CA. – 1985. – P. 1076–1078.
12. *Fulkerson D.*, *Gross O.* Incidence matrices and interval graphs // Pacific J. Math. – 1965. – **15**(3). – P. 835–855.
13. *Gogate V.*, *Dechter R.* Approximate counting by sampling the backtrack-free search space // Proc. of AAAI-07. Vancouver, CA. – 2007. – P. 198–203.
14. *Gogate V.*, *Dechter R.* Approximate Solution Sampling (and Counting) on AND/OR search spaces // Proc. of CP-08. Sydney, AU. – 2008. – P. 534–538.
15. *Gomes C.*, *Sabharwal A.*, *Selman B.* Model counting: A new strategy for obtaining good bounds // Proc. of AAAI-06. Boston, MA. – 2006. – P. 534–538.
16. *From* sampling to model counting / C. Gomes, J. Hoffmann, A. Sabharwal et al. // Proc. of IJCAI-07. Hyderabad, India. – 2007. – P. 2293–2299.
17. *Counting* CSP solutions using generalized XOR constraints / C. Gomes,W-J. van Hoeve, A. Sabharwal et al. // Proc. of AAAI-07. Vancouver, BC. – 2007. – P. 204–209.
18. *Gomes C.*, *Sabharwal A.*, *Selman B.* Handbook of Satisfiability, chapter 20, Model Counting. IOS Press, 2009.
19. *Jégou P.*, *Terrioux C.* Hybrid backtracking bounded by tree-decomposition of constraint networks // Artificial Intelligence. – 2003. – **146**. – P. 43–75.
20. *Kask K.*, *Dechter R.*, *Gogate V.* New look-ahead schemes for constraint satisfaction // Proc. of AI&M, 2004.
21. *Kroc L.*, *Sabharwal A.*, *Selman B.* Leveraging belief propagation, backtrack search, and statistics for model counting // Proc. of CPAIOR-08. Paris, France. – 2008. – P. 127–141.

End on the page 70

22. *Kumar T.* A model counting characterization of diagnoses // Proc. of the 13th International Work-shop on Principles of Diagnosis, 2002.
23. *Last* conflict based reasoning / C. Lecoutre, L. Sais, S. Tabary, V. Vidal // Proc. of ECAI-06. Trento, Italy. – 2006. – P. 133–137.
24. *Mann M.*, *Tack G.*, *Will S.* Decomposition during search for propagation-based constraint solvers. CoRR, abs/0712.2389, 2007.
25. *Montanari U.* Network of constraints: Fundamental properties and applications to picture processing // Inf. Sci. – 1974. – N 7. – P. 95–132.
26. *Nishimura N.*, *Ragde P.*, *Szeider S.* Solving #SAT using vertex covers // Acta Inf. – 2007. – **44**(7). – P. 509–523.
27. *Pesant G.* Counting solutions of CSPs: A structural approach // Proc. of IJCAI-05. Edinburgh, Scotland. – 2005. – P. 260–265.
28. *Robertson N.*, *Seymour P.* Graph minors II: Algorithmic aspects of tree-width // Algorithms. – 1986. – N 7. – P. 309–322.
29. *Rose D.* Tringulated graphs and the elimination process // J. of Mathematical Analysis and its Applications. – 1970. – N 32.
30. *Roth D.* On the hardness of approximate reasoning // Artificial Intelligence. – 1996. – **82**(1–2). – P. 273–302.
31. *Samer M.*, *Szeider S.* Algorithms for propositional model counting // J. of Discrete Algorithms. – 2010. – N 8. – P. 50–64.
32. *Combining* component caching and clause learning for effective model counting / T. Sang, F. Bacchus, P. Beame et al. // In SAT-04, Vancouver, Canada, 2004.
33. *Thurley M.* SharpSAT – counting models with advanced component caching and implicit BCP // Proc. of SAT-06. Seattle, WA. – 2006. – P. 424–429.
34. *Valiant L.* The complexity of computing the permanent // Theoretical Computer Sciences. – 1979. – N 8. – P. 189–201.
35. *Wei W.*, *Selman B.* A new approach to model counting // Proc. of SAT-05. St. Andrews, UK. – 2005. – P. 324–339.
36. *Zanarini A.*, *Pesant G.* Solution counting algorithms for constraint-centered search heuristics // Constraints. – 2009. – **14**(3). – P. 392–413.

●