Exploiting Problem Structure for Solution Counting

Aurélie Favier[1], Simon de Givry[1], and Philippe Jégou[2]

[1] INRA MIA Toulouse, France {afavier,degivry}@toulouse.inra.fr
[2] Université Paul Cézanne, Marseille, France philippe.jegou@univ-cezanne.fr

Abstract. This paper deals with the challenging problem of counting the number of solutions of a CSP, denoted #CSP. Recent progress have been made using search methods, such as BTD [15], 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 sparse constraint graphs but 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 shows the practical efficiency of our proposed approaches.

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 [27]. This problem has numerous applications in computer science, particularly in AI, e.g. in approximate reasoning [23], in diagnosis [18], in belief revision [5], *etc.*

In the literature, two principal classes of approaches have been proposed. The first class, methods find exactly the number of solutions. The second class, methods propose approximations. 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 [6] and AND/OR graph search [8, 9] for counting. We propose to adapt Backtracking with Tree-Decomposition (BTD) [15] to #CSP. This method was initially proposed for solving structured CSPs. Our modifications to BTD are similar to what has been done in the AND/OR context [8, 9], 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 #SAT, the model counting problem associated with SAT [27]. Exact methods for #SAT extend systematic SAT solvers, adding component analysis [3] and caching [26] for efficiency. Approaches using approximations estimate the number of solutions.

They propose poly-time or exponential time algorithms which must offer reasonably good approximations of the number of solutions, with theoretical guarantees about the quality of the approximation, or not. Again, most of the work has been done on #SAT by sampling either the original OR search space [28, 12, 10, 17], or the original AND/OR search space [11]. All these methods except that in [28] 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 approach involves reducing the search space by adding streamlining XOR constraints [13, 14]. However, it does not guarantee that the resulting problem is easier to solve.

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 our modified BTD. This task should be relatively easy if the original instance has a sparse graph[3]. Finally, an approximate number of solutions on the whole problem is obtained by combining the results of each subproblem. The resulting approximate method called `ApproxBTD` 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 [16], or in the related context of Bayesian inference, iterative belief propagation and the edge deletion framework [4][4]. These approaches do not exploit the local structure of 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 tree-decomposition. Section 3 describes BTD for counting and Section 4 presents ApproxBTD for approximate counting. Experimental results are given in Section 5, then we conclude.

2 Preliminaries

A CSP is a quadruplet $\mathcal{P} = (X, D, C, R)$. X and D are sets of n variables and finite domains. The domain of variable x_i is denoted d_{x_i}. The maximum domain size is d. C is a set of m constraints. Each constraint $c \in C$ is a set $\{x_{c_1}, \ldots, x_{c_k}\}$ of variables. A relation $r_c \in R$ is associated with each constraint c such that r_c represents (in intension) the set of allowed tuples over $d_{x_{c_1}} \times \cdots \times d_{x_{c_k}}$. An *assignment* $Y = \{x_1, \ldots, x_k\} \subseteq X$ is a tuple $\mathcal{A} = (v_1, \ldots, v_k)$ from $d_{x_1} \times \cdots \times d_{x_k}$. A constraint c is said *satisfied* by $\mathcal{A}$ if $c \subseteq Y, \mathcal{A}[c] \in r_c$, *violated* otherwise. A solution is a complete assignment 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.

[3] In fact, it depends on the tree-width of the subproblems, which is bounded by the maximum clique size of the original instance. In the case of a sparse graph, we expect this size to be small. This forbids using our approach for solution counting in CSPs with global constraints.

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

A *tree-decomposition* [22] 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 $\mathcal{C}=\{C_i : i \in I\}$ a family of subsets of X, such that each cluster C_i is a node of $\mathcal{T}$ and satisfies: (1) $\cup_{i\in I} C_i = X$, (2) for each constraint $c \in C$, there exists $i \in I$ with $c \subseteq C_i$, (3) for all $i,j,k \in I$, if k is on a path from i to j in $\mathcal{T}$, then $C_i \cap C_j \subseteq C_k$. The width of a tree-decomposition $(\mathcal{C},\mathcal{T})$ is equal to $max_{i\in I}|C_i|-1$. The tree-width of $\mathcal{P}$ is the minimum width over all its tree-decompositions. Finding an optimal tree-decomposition is NP-Hard [2]. In the following, from a tree-decomposition, we consider a rooted tree (I,F) with root C_1 and we note $Sons(C_i)$ the set of son clusters of C_i and $Desc(C_j)$ the set of variables which belong to C_j or to any descendant C_k of C_j in the subtree rooted in C_j.

3 Exact solution counting with BTD

The essential property of tree decomposition is that assigning $C_i \cap C_j$ (C_j is a son of C_i) separates the initial problem into two subproblems, which can then be solved independently. The first subproblem rooted in C_j is defined by the variables in $Desc(C_j)$ and by all the constraints involving *at least* one variable in $Desc(C_j) \setminus C_i$. The remaining constraints, together with the variables they involve, define the remaining subproblem.

A tree search algorithm can exploit this property by using a suitable variable ordering : the variables of any cluster C_i must be assigned before the variables that remain in its son clusters. In this case, for any cluster $C_j \in Sons(C_i)$, once $C_i \cap C_j$ is assigned, the subproblem rooted in C_j conditioned by the current assignment $\mathcal{A}$ of $C_i \cap C_j$ can be solved independently of the rest of the problem. The exact number of solutions *nb* of this subproblem may then be recorded, called a #good and represented by a pair $(\mathcal{A}[C_i \cap C_j], nb)$, which means it will never be computed again for the same assignment of $C_i \cap C_j$. This is why algorithms such as BTD or AND / OR graph search are able to keep the complexity exponential in the size of the largest cluster only.

BTD is described in Algorithm 1. Given an assignment $\mathcal{A}$ and a cluster C_i, BTD looks for the number of extensions $\mathcal{B}$ of $\mathcal{A}$ on $Desc(C_i)$ such that $\mathcal{A}[C_i\text{-}V_{C_i}] = \mathcal{B}[C_i\text{-}V_{C_i}]$. V_{C_i} denotes the set of unassigned variables of C_i. The first call is to BTD($\emptyset, C_1, C_1$) and it returns the number of solutions. Inside a cluster C_i, it proceeds classically by assigning a value to a variable and by backtracking if any constraint is violated. When every variable in C_i is assigned, BTD computes the number of solutions of the subproblem induced by the first son of C_i, if there is one. More generally, let us consider C_j, a son of C_i. Given a current assignment $\mathcal{A}$ on C_i, BTD checks whether the assignment $\mathcal{A}[C_i \cap C_j]$ corresponds to a #good. If so, BTD multiplies the recorded number of solutions with the number of solutions of C_i with $\mathcal{A}$ as its assignment. Otherwise, it extends $\mathcal{A}$ on $Desc(C_i)$ in order to compute its number of consistent extensions *nb*. Then, it records the #good $(\mathcal{A}[C_i \cap C_j], nb)$. BTD computes the number of solutions of the subproblem induced by the next son of C_i. Finally, when each son of C_i has been examined, BTD tries to modify the current assignment of C_i. The number of solutions of C_i is the sum of solution counts for every assignment of C_i. The time (resp. space) complexity of BTD for #CSP is the same as for CSP: $O(n.m.d^{w+1})$ (resp. $O(n.s.d^s)$) with $w+1$ the size of the largest C_k and s the size of the largest intersection $C_i \cap C_j$ (C_j is a son of C_i). In practice, for problems with large tree-width, BTD runs out of time and memory, as shown in Section 5. In this case, we are interested in an approximate method.

Algorithm 1: BTD($\mathcal{A}$, C_i, V_{C_i}) : integer

```
if V_Ci = ∅ then
    if Sons(C_i) = ∅ then return 1;
    else
        S ← Sons(C_i) ; NbSol ← 1 ;
        while S ≠ ∅ and NbSol ≠ 0 do
            choose C_j in S ; S ← S − {C_j};
            if (A[C_i ∩ C_j], nb) is a #good in P then NbSol ← NbSol × nb; else
                nb ← BTD(A, C_j, V_Cj − (C_i ∩ C_j));
                record #good (A[C_i ∩ C_j], nb) of C_i/C_j in P;
                NbSol ← NbSol × nb;
        return NbSol;
else
    choose x ∈ V_Ci ; d ← d_x ; NbSol ← 0 ;
    while d ≠ ∅ do
        choose v in d ; d ← d-{v};
1       if A ∪ {x ← v} does not violate any c ∈ C then
            NbSol ← NbSol+BTD(A ∪ {x ← v}, C_i, V_Ci-{x});
    return NbSol;
```

4 Approximate solution counting with ApproxBTD

We consider here CSPs that are not necessarily structured. We can define a collection of subproblems of a CSP by partitioning the set of constraints, that is the set of edges in the graph. We remark that each graph (X,C) can be partitioned into k subgraphs (X_1,E_1), $\ldots$, (X_k,E_k), such that $\cup X_i = X$, $\cup E_i = C$ and $\cap E_i = \emptyset$, and such that each (X_i,E_i) is chordal[5]. So, each (X_i,E_i) can be associated to a structured subproblem $\mathcal{P}_i$ (with corresponding sets of variables X_i and constraints E_i), which can be efficiently solved using BTD. Assume that $\mathcal{S}_{\mathcal{P}_i}$ is the number of solutions for each subproblem $\mathcal{P}_i$, $1 \leqslant i \leqslant k$. We will estimate the number of solutions of $\mathcal{P}$ exploiting the following property. We denote $\mathbb{P}_{\mathcal{P}}(\mathcal{A})$ the probability of "$\mathcal{A}$ *is a solution of* $\mathcal{P}$" . $\mathbb{P}_{\mathcal{P}}(\mathcal{A}) = \dfrac{\mathcal{S}_{\mathcal{P}}}{\prod_{x\in X} d_x}$.

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.*

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

Notice that the approximation returns an exact answer if all the subproblems are independent ($\cap X_i = \emptyset$) or $k = 1$ ($\mathcal{P}$ is already chordal) or if there exists an inconsistent subproblem $\mathcal{P}_i$. 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 [21] to construct an upper bound).

$$\mathcal{S}_{\mathcal{P}} \leq \min_{i\in[1,k]} \left\lceil \frac{\mathcal{S}_{\mathcal{P}_i}}{\prod_{x\in X_i} d_x} \times \prod_{x\in X} d_x \right\rceil$$

[5] 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.

Algorithm 2: ApproxBTD($\mathcal{P}$) : integer

Let $G' = (X', C')$ be the constraint graph associated with $\mathcal{P}$;
$i \leftarrow 0$;
while $G' \neq \emptyset$ **do**
  $i \leftarrow i + 1$;
  Compute a partial chordal subgraph (X_i, E_i) of G' ;
  Let $\mathcal{P}_i$ be the subproblem associated with (X_i, E_i) ;
  $\mathcal{S}_{\mathcal{P}_i} \leftarrow$BTD($\emptyset, C'_1, C'_1$) with C'_1 the root cluster of the tree-decomposition of $\mathcal{P}_i$;
  $G' \leftarrow (X', C' - E_i)$ with X' be the set of variables induced by C'-E_i ;
$k \leftarrow i$;
return $\left\lceil \prod_{i=1}^{k} \frac{\mathcal{S}_{\mathcal{P}_i}}{\prod_{x \in X_i} d_x} \times \prod_{x \in X} d_x \right\rceil$;

Our proposed method called ApproxBTD 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 $1 \leqslant i \leqslant k$. Subproblems associated to (X_i, E_i) are solved with BTD. The method returns an approximation to the number of solutions of $\mathcal{P}$ using Property 1.

The number of iterations is less than n (at least we have $n-1$ edges (a tree) at each iteration or vertices have been deleted). Each chordal subgraph and its associated optimal tree-decomposition can be computed in $O(nm)$ [7][6]. Moreover, we guarantee that the tree-width w (plus one) of each computed chordal subgraph is at most equal to K, the maximum clique size (the *clique number*) of $\mathcal{P}$. Let w^* be the tree-width of $\mathcal{P}$, we have $w+1 \leq K \leq w^*+1$. Finally, the time complexity of ApproxBTD is $O(n^2 m d^K)$.

5 Experimental results

We implemented BTD and ApproxBTD counting methods on top of `toulbar2` C++ solver[7]. The experimentations were performed on a 2.6 GHz Intel Xeon computer with 4GB running Linux 2.6.27-11-server. Reported times are in seconds. We limit to one hour the time spent for solving a given instance ('-' time out, 'mem' memory out). In BTD (line 1), 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 [19], is used inside clusters. Our methods exploit a binary branching scheme. The variable is assigned to its first value or this value is removed from the domain.

We performed experiments on SAT and CSP benchmarks[8]. 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) and genotype analysis in complex pedigrees [25]. This problem involves counting the number of consistent genotype configurations satisfying genotyping partial observa-

[6] It returns a maximal subgraph for binary CSP. For non-binary CSP, we do not guarantee subgraph maximality and add to the subproblem all constraints totally included in the subgraph.
[7] http://mulcyber.toulouse.inra.fr/projects/toulbar2 version 0.8.
[8] From www.satlib.org, www.satcompetition.org and mat.gsia.cmu.edu/COLOR02/.

tions and Mendelian laws of inheritance. The corresponding decision problem is NP-complete [1]. We selected instances from [25], removing genotyping errors beforehand.

We compared BTD with state-of-the-art #SAT solvers `Relsat` [3] v2.02 and `Cachet` [26] v1.22, and also `c2d` [6] v2.20 which also exploits the problem structure. Both methods uses *MinFill* variable elimination ordering heuristic (except for *hanoi* where we used the default file order) to construct a tree-decomposition / d-DNNF. We also compared ApproxBTD with approximation method `SampleCount` [12]. 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`. 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 a domain value is selected, and a set of binary clauses to forbid multiple value selection).

The following table summarizes our results. The columns are : instance name, number of variables, number of constraints / clauses, width of the tree-decomposition, exact number of solutions if known, time for `c2d`, `Cachet`, `Relsat`, and BTD; for ApproxBTD : maximum tree-width for all chordal subproblems, approximate number of solutions, and time; and for `SampleCount-A` and `SampleCount-LB` : approximate number of solutions, corresponding upper bound, and time. We reported total CPU times as given by c2d, cachet, relsat (precision in seconds). For BTD and ApproxBTD, the total time does not include the task of finding a variable elimination ordering. For SampleCount, we reported total CPU times with the bash command "time". We noticed that BTD can solve instances with relatively small tree-width (except for *le450* which has few solutions). Exact #SAT solvers generally perform better than BTD on SAT instances (except for *hanoi5*) but have difficulties on translated CSP instances. Here, BTD maintaining arc consistency performed better than #SAT solvers using unit propagation. Our approximate method ApproxBTD exploits a partition of the constraint graph in such a way that the resulting subproblems to solve have a small tree-width ($w \leq 11$) on these benchmarks. It has the practical effect that the method is relatively fast whatever the original tree-width. The quality of the approximation found by ApproxBTD is relatively good and it is comparable (except for ssa and logistics benchmarks) to `SampleCount`, which takes more time. For graph coloring, BTD and ApproxBTD outperform also a dedicated CSP approach (*2_Insertion_3* $\geq$ 2.3e12, *mug100_1* $\geq$ 1.0e28 and *games120* $\geq$ 4.5e42 in 1 minute each; *myciel5* $\geq$ 4.1e17 in 12 minutes, times were measured on a 3.8GHz Xeon as reported in [14]).

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 small tree-width. For problems with large tree-width and sparse constraint graph, we have presented a new approximate method whose quality is comparable with existing methods and which is much faster than other approaches and which requires no parameter tuning (except the choice of a tree decomposition heuristic). Other structural parameters [20, 24] should deserve future work.

Instances	Vars	w	Solutions	c2d	cachet	relsat	BTD	ApproxBTD			SampleCount-A		SampleCount-LB	
	(Bool vars)			Time	Time	Time	Time	w	Solutions	Time	Solutions	Time	Solutions	Time
SAT														
wff.3.100.150	100	39	1.8e21	-	-	-	mem	2	$\approx$ **2.21e21**	$\leq$ 1.95e27	0.02	$\approx$ 1.37e21	959.8	-	-
wff.3.150.525	150	92	1.4e14	-	-	**2509**	mem	2	$\approx$ 9.93e14	$\leq$ 7.80e40	0.2	$\approx$ **3.80e14**	0.68	$\geq$ 2.53e12	4.65
ssa7552-038	1501	25	2.84e40	**0.15**	0.22	67	0.72	5	$\approx$ 2.47e37	$\leq$ 1.51e138	1.05	$\approx$ **1.11e40**	134.2	$\geq$ 3.54e38	1162.06
ssa7552-160	1391	12	7.47e32	0.12	**0.08**	5	0.29	5	$\approx$ 1.56e34	$\leq$ 2.23e113	0.82	$\approx$ **5.08e32**	144.6	$\geq$ 2.31e31	1293
2bitcomp_5	125	36	9.84e15	0.47	**0.15**	1	11.53	5	$\approx$ 9.50e16	$\leq$ 1.75e31	0.03	$\approx$ **4.37e15**	0.184	$\geq$ 3.26e15	1.23
2bitmax_6	252	58	2.10e29	18.71	**1.57**	20	mem	5	$\approx$ 2.69e30	$\leq$ 4.27e65	0.14	$\approx$ **1.62e29**	1.676	$\geq$ 2.36e26	10.34
ais10	181	116	296	17.14	29.31	**6**	390.32	9	$\approx$ 1	$\leq$ 2.86e22	1.05	$\approx$ **124**	45.93	$\geq$ 20	312.74
ais12	265	181	1328	1162.75	2169	**229**	-	11	$\approx$ **1**	$\leq$ 1.64e40	3.78	$\approx$ 0	9.156	$\geq$ 0	9.17
logistics.a	828	116	3.8e14	-	**3.82**	10	mem	10	$\approx$ 1	$\leq$ 2.33e147	13.30	$\approx$ **7.25e11**	170.9	$\geq$ 0	605.33
logistics.b	843	107	2.3e23	-	**12.4**	433	mem	13	$\approx$ 1	$\leq$ 2.28e143	13.72	$\approx$ **2.13e23**	198.7	$\geq$ 0	229.86
hanoi4	718	46	1	3.41	32.69	3	**1.72**	6	$\approx$ **1**	$\leq$ 8.65e107	1.57	$\approx$ 0	5.24	$\geq$ 0	5.25
hanoi5	1931	58	1	-	-	-	**25.46**	7	$\approx$ **1**	$\leq$ 2.62e298	15.69	$\approx$ 0	6.14	$\geq$ 0	6.16
coloring														
2-Insertions_3	37 (148)	9	6.84e13	235	-	-	**7.9**	1	$\approx$ **1.91e13**	$\leq$ 6.00e17	0.01	$\approx$ 4.73e12	1.00	$\geq$ 4.73e12	7.42
2-Insertions_4	149 (596)	38	-	-	-	-	-	1	$\approx$ 1.30e22	$\leq$ 1.64e71	0.07	$\approx$ 0	3.76	$\geq$ 0	3.78
DSJC125.1	125 (625)	65	-	-	-	-	-	3	$\approx$ 1.23e13	$\leq$ 2.27e70	0.13	$\approx$ 0	73.14	$\geq$ 0	73.17
games120	120 (1080)	41	-	-	-	-	-	8	$\approx$ 1.12e78	$\leq$ 1.92e99	9.83	$\approx$ 0	13.76	$\geq$ 1.35e61	91.07
GEOM30a	30 (180)	6	4.98e14	0.86	-	-	**0.08**	5	$\approx$ **7.29e14**	$\leq$ 1.81e15	0.03	$\approx$ 1.23e13	0.432	$\geq$ 3.28e12	3.70
GEOM40	40 (240)	5	4.1e23	1	-	-	**0.09**	5	$\approx$ **4.42e23**	$\leq$ 1.10e24	0.02	$\approx$ 2.14e20	1.552	$\geq$ 6.50e19	9.31
le450_5a	450 (2250)	315	3840	-	343.68	**326**	953.24	4	$\approx$ **1**	$\leq$ 2.41e216	2.87	$\approx$ 0	8.56	$\geq$ 0	8.58
le450_5b	450 (2250)	318	120	-	242.30	**187**	1167.78	4	$\approx$ **1**	$\leq$ 5.71e216	2.90	$\approx$ 0	8.59	$\geq$ 0	8.61
le450_5c	450 (2250)	315	120	-	**20.79**	57	43.66	4	$\approx$ **1**	$\leq$ 1.49e201	6.65	$\approx$ 0	110.5	$\geq$ 0	110.56
le450_5d	450 (2250)	299	960	-	**16.07**	36	85.03	4	$\approx$ **1**	$\leq$ 8.58e200	6.64	$\approx$ 0	54.57	$\geq$ 0	54.58
mug100_1	100 (400)	3	1.3e37	0.19	-	-	**0.02**	2	$\approx$ **5.33e37**	$\leq$ 7.08e41	0.01	$\approx$ 4.2e34	2.08	$\geq$ 4.20e34	15.59
myciel5	47 (282)	21	-	-	-	-	mem	1	$\approx$ 7.70e17	$\leq$ 8.53e32	0.02	$\approx$ 7.29e17	0.86	$\geq$ 7.29e17	6.37
pedigree														
parkinson	34 (340)	4	3.56e10	31.34	5.33	344	**0.12**	2	$\approx$ 2.33e10	$\leq$ 6.3e11	0.02	$\approx$ **4.08e10**	3.58	$\geq$ 9.17e8	25.62
moissac3	72 (1350)	2	4.89e35	0.98	0.09	$<$ 1	**0.02**	2	$\approx$ **3.40e35**	$\leq$ 6.11e35	0.02	$\approx$ 1.18e34	16.93	$\geq$ 6.75e31	118.41
langladeM7	427 (8818)	8	6.73e196	-	-	-	**31.79**	2	$\approx$ **3.81e193**	$\leq$ 8.07e203	0.14	$\approx$ 3.03e179	816.9	-	-

References

1. L. Aceto, J. A. Hansen, A. Ingólfsdóttir, J. Johnsen, and J. Knudsen. The complexity of checking consistency of pedigree information and related problems. *Journal of Computer Science Technology*, 19(1):42–59, 2004.
2. S. Arnborg, D. Corneil, and A. Proskurowski. Complexity of finding embeddings in a k-tree. *SIAM Journal of Discrete Mathematics*, 8:277–284, 1987.
3. R. Bayardo and J. Pehoushek. Counting models using connected components. In *AAAI-00*, pages 157–162, 2000.
4. A. Choi and A. Darwiche. An edge deletion semantics for belief propagation and its practical impact on approximation quality. In *Proc. of AAAI*, pages 1107–1114, 2006.
5. A. Darwiche. On the tractable counting of theory models and its applications to truth maintenance and belief revision. *Journal of Applied Non-classical Logic*, 11:11–34, 2001.
6. A. Darwiche. New advances in compiling cnf to decomposable negation normal form. In *Proc. of ECAI*, pages 328–332, 2004.
7. P.M Dearing, D.R. Shier, and D.D. Warner. Maximal chordal subgraphs. *Discrete Applied Mathematics*, 20(3):181–190, 1988.
8. Rina Dechter and Robert Mateescu. The impact of and/or search spaces on constraint satisfaction and counting. In *Proc. of CP*, pages 731–736, Toronto, CA, 2004.
9. Rina Dechter and Robert Mateescu. And/or search spaces for graphical models. *Artif. Intell.*, 171(2-3):73–106, 2007.
10. Vibhav Gogate and Rina Dechter. Approximate counting by sampling the backtrack-free search space. In *Proc. of AAAI-07*, pages 198–203, Vancouver, CA, 2007.
11. Vibhav Gogate and Rina Dechter. Approximate solution sampling (and counting) on and/or search spaces. In *Proc. of CP-08*, pages 534–538, Sydney, AU, 2008.
12. Carla P. Gomes, Joerg Hoffmann ans Ashish Sabharwal, and Bart Selman. From sampling to model counting. In *Proc. of IJCAI*, pages 2293–2299, 2007.
13. Carla P. Gomes, Ashish Sabharwal, and Bart Selman. Model counting: A new strategy for obtaining good bounds. In *Proc. of AAAI-06*, Boston, MA, 2006.
14. Carla P. Gomes, Willem-Jan van Hoeve, Ashish Sabharwal, and Bart Selman. Counting CSP solutions using generalized XOR constraints. In *Proc. of AAAI-07*, pages 204–209, Vancouver, BC, 2007.
15. P. Jégou and C. Terrioux. Hybrid backtracking bounded by tree-decomposition of constraint networks. *Artificial Intelligence*, 146:43–75, 2003.
16. Kalev Kask, Rina Dechter, and Vibhav Gogate. New look-ahead schemes for constraint satisfaction. In *Proc. of AI&M*, 2004.
17. Lukas Kroc, Ashish Sabharwal, and Bart Selman. Leveraging belief propagation, backtrack search, and statistics for model counting. In *Proc. of CPAIOR-08*, pages 127–141, Paris, France, 2008.
18. T.K Satish Kumar. A model counting characterization of diagnoses. In *Proc. of the 13th International Workshop on Principles of Diagnosis*, 2002.
19. C. Lecoutre, L. Sais, S. Tabary, and V. Vidal. Last conflict based reasoning. In *Proc. of ECAI-2006*, pages 133–137, Trento, Italy, 2006.
20. Naomi Nishimura, Prabhakar Ragde, and Stefan Szeider. Solving #sat using vertex covers. *Acta Inf.*, 44(7):509–523, 2007.
21. Gilles Pesant. Counting solutions of CSPs: A structural approach. In *Proc. of IJCAI*, pages 260–265, 2005.
22. N. Robertson and P.D. Seymour. Graph minors II: Algorithmic aspects of tree-width. *Algorithms*, 7:309–322, 1986.

23. D. Roth. On the hardness of approximate reasonning. *Artificial Intelligence*, 82(1-2):273–302, 1996.
24. Marko Samer and Stefan Szeider. A fixed-parameter algorithm for #sat with parameter incidence treewidth, 2006.
25. M. Sanchez, S. de Givry, and T. Schiex. Mendelian error detection in complex pedigrees using weighted constraint satisfaction techniques. *Constraints*, 13(1):130–154, 2008.
26. T. Sang, F. Bacchus, P. Beame, H. Kautz, and T. Pitassi. Combining component caching and clause learning for effective model counting. In *SAT-04*, Vancouver, Canada, 2004.
27. L. G. Valiant. The complexity of computing the permanent. *Theoretical Computer Sciences*, 8:189–201, 1979.
28. Wei Wei and Bart Selman. A new approach to model counting. In *Proc. of SAT-05*, pages 324–339, St. Andrews, UK, 2005.