Scribe Notes of October 15, 2009
Speaker: Shant
Scribe: Chris

Paper: *Exploiting Problem Structure for Solution Counting*
Authors: Aurélie Favier, Simon de Givry, and Philippe Jégou

1. Summary of idea & main results

The problem & the proposed approaches. This paper addresses the problem of counting the number of solutions of a CSP, which is the #CSP problem. There are two possible approaches to this problem:

1. *Exact methods*. As an exact approach, the authors propose to use the Backtracking Tree Decomposition (BTD) algorithm, which exploits the structure of the problem. Below, we choose to refer to this technique as #BTD to differentiate it from BTD.
2. *Approximate methods*. As an approximate method, they propose a new algorithm, called ApproxBTD, which is based on applying the BTD to a decomposition of the original CSP into subproblems.

Motivation. Because, in practice, the number of solutions of a CSP can be huge, it is not always practical to use exact techniques. In those situations, approximate techniques become highly desirable. The authors recommend using the #BTD when seeking the exact number of solutions. Because it exploits the structure of the problem, this technique can be significantly better than enumeration techniques. When it is not practical to compute the exact number of solutions, the authors propose ApproxBTD, which also exploits the problem structure. ApproxBTD provides a good approximation of the number of solutions as well as an upper bound on the number of solutions.

The central idea. The key idea in the paper is to compute the number of solutions of a given CSP by finding the product of the number of solutions of a set of subproblems obtained from the original problem by decomposition. The decomposition exploits the structure of the problem:

1. The exact method, #BTD, exploits the idea that, in a tree decomposition of the CSP, the number of solutions of the CSP can be computed *recursively* by multiplying the number solutions of the children of the subproblem for each solution of the root subproblem then summing the numbers obtained for each solution of the root problem. You can multiply the solutions for each child node because they are truly separate solutions.
2. The approximate methods, ApproxBTD, exploits the idea that a CSP problem can be decomposed into a number of subproblems whose constraint graphs are chordal and maximal (MaxChord algorithm). The resulting subproblems typically have a low tree width, which allows us to 'efficiently' compute their sizes (using #BTD) to then compute the approximate number of solutions in the original problem as well as an upper bound.

The main results. The authors show:

1. How the BTD can be used to compute the exact number of solutions using the structure of the tree decomposition. They empirically demonstrate that it is competitive against known

techniques when the tree width of the chosen tree decomposition of the CSP (which is found by tree clustering) is small.

2. How ApproxBTD can be used to compute the approximate numbers of solutions, as well as, an upper bound on the number of solutions. They empirically demonstrate that ApproxBTD is quicker than competing approximation techniques and gives good approximation results.

2 BTD and #BTD

Tree decomposition of a CSP. Shant recalled the definition of a tree decomposition and that of the 'tree width' parameter:

- The *tree width of a tree decomposition* is a parameter that characterizes a given tree obtained by a given tree decomposition technique. It is the number of variables in the largest cluster in the tree minus one.
- The *tree width of a CSP* is a parameter that characterizes the constraint network of the CSP (i.e., the parameter does not depend on the particular tree decomposition technique used). It can be computed as the minimum tree width of all possible tree decompositions of the CSP, and finding it is NP-hard.

Solving a CSP with BTD. In a tree decomposition, a *separator* is a set of variables that are common between two adjacent tree nodes. They are called separators because removing them from the problem (or instantiating them) will separate the two tree nodes. BTD proceeds by solving the problem from the root of the tree down to the leaves. It uses backtrack search to solve a given tree node. As partial solutions are propogated from the root to a leaf node (respectively, a descendant node) and are found to be consistent (respectively, inconsistent), the projection of the solutions on the separators are recorded as lists of goods (respectively, no-goods). When the same values for the separators are explored in another solution starting from the root, the search is interrupted because it is known that the path will yield a solution (respectively, a dead-end). In this way you can reuse the goods (and no-goods) in future assignments to save work. This is done using the following algorithm:

1. Solve the first/root problem
2. Assign values to the variables in the root problem which appear in a solution of the problem.
3. For each child problem, assign the variables in the separator the values that they were given in the parent problem.
4. Extend the solution to the child
    - If this succeeds, you can record a good for the values given to the variables in the separator
    - If this fails, you can record a no-good for the values given to the variables in the separator

In this way you build the solution top down with a backtrack search. The problem with this approach is that the separators can be large and the list of goods and no-goods can grow to use all of the memory.

#BTD: Exact solution counting using the BTD. BTD is modified to perform exact solution counting in the #BTD algorithm. Instead of recording goods/no-goods the number of solutions is recorded. Otherwise, the process proceeds as for described above.

3 ApproxBTD

Problem decomposition. ApproxBTD uses the MaxChord algorithm to decompose the CSP into independent subproblems:

- Importantly, the constraints are *partitioned* across the subproblems: each constraint appears in exactly one subproblem.
- Each variable is in at least one partition, but may appear in more than one.
- Each subproblem is chordal

Approximate Solution Counting: ApproxBTD. Below, we call the size of a CSP the product of the sizes of its domain. The authors introduce the following:

- The probability of an assignment of all the variables of a subproblem being a solution in a subproblem is the ratio of the number of solutions in the subproblem (computed using #BTD) over the size of the subproblem.
- Assuming that the subproblems are independent (which is an approximation and does not always hold because the same variable may appear in more than one subproblem), the probability of a solution in the CSP is obtained by multiplying the probabilities of a solution in all subproblems.
- The number of solution of the CSP can be multiplying the probability of a solution in the CSP by the size of the CSP.
- An upper bound on the number of solutions of the CSP is obtained by multiplying the smallest probability of a solution to a subproblem by the size of the original CSP.

Shant explained the meaning of the above parameters and motivated them with examples.

4 Discussion

Some of the questions asked were as follows:

Q: What applications can benefit from counting the number of solutions?

A: The authors identify three application areas (model counting, diagnostic, etc.) where solution counting is important. Dechter talked about her work on inference in Bayed nets during her visit to UNL, but we have not studied that in details. In modeling a real world, imagine a situation where the number of solutions of a model can be related to the quality of the model. You may have several models and you want find the number of solutions to each model to get an approximation of the quality of the model. Also, in diagnostics, if you observe a certain behavior it is useful to know how many possibilities this behavior entails.

Q: Is there a better way to get an upper bound by pre-processing the CSP in some way?

A: Yes, running arc consistency or some other local consistency method before creating the tree decomposition may eliminate some inconsistent solutions.

Comment: The MaxChord [Dearing et al., 88] is an interesting (but complicated) paper to study.