Pseudo-Tree Search with Soft Constraints

Javier Larrosa[1] and **Pedro Meseguer**[2] and **Martí Sánchez**[3]

Abstract. Pseudo-tree search is a well known algorithm for CSP solving. It exploits the problem structure to detect independent subproblems that are solved separately. Its main advantage is that its run time complexity is bounded by a problem structural parameter. In this paper, we extend this idea to soft constraint problems. We show that the same general principles apply to this domain. However, a naive implementation is not competitive with state-of-the-art algorithms, because solving independent problems separately may yield a poor algorithmic efficiency due to loose upper bounds. We introduce PT-BB, a branch-and-bound algorithm that performs efficient pseudo-tree search. Its main feature is the use of local upper bounds which can improve over loose global upper bounds. We also show that PT-BB combines nicely with russian doll search (RDS), producing an interesting algorithm.

1 INTRODUCTION

Constraint satisfaction problems (CSPs) involve the assignment of *values* to *variables*, subject to a set of *constraints*. Many interesting problems can be modelled as CSPs. Solving techniques can be roughly divided into *search* and *decomposition methods*. The main advantage of search is its polynomial space complexity. Its main disadvantage is its time complexity, exponential on the number of variables. The main advantage of decomposition methods is their time complexity, exponential on a topological parameter called *width*. The width is always less than or equal to the number of variables and some important problems have small width. The main disadvantage is the space complexity, also exponential on the width. The high space complexity makes these methods impractical in many cases.

Pseudo-tree search, introduced by Freuder and Quinn for CSP [4], is a search algorithm that exploits the problem structure to make search more efficient. Search is conducted over a pseudo-tree arrangement of the problem which allows the detection of independent subproblems that are solved separately. The algorithm inherits from search methods its polynomial space complexity. It also has, similarly to decomposition methods, a bounded time complexity $O(exp(h))$, where h is the height of the pseudo-tree arrangement. Interestingly, Bayardo and Miranker [1] showed that h is no more than a logarithm away of the width used by decomposition methods.

In the last years, the CSP framework has been augmented with the so-called *soft constraints* with which it is possible to express preferences among solutions [9, 3]. Soft constraint frameworks associate costs to tuples and the goal is to find a complete assignment with minimum combined cost. Therefore, soft constraints specify optimization problems, which increases the expressivity of the framework and therefore, its applicability.

In this paper we extend pseudo-tree search to soft constraint problems. For simplicity reasons, we will develop our work for *weighted* CSPs (WCSPs), where costs are natural numbers and global costs are computed by summing partial costs. The extension to other soft-constraint frameworks is direct. We introduce *pseudo-tree branch-and-bound* (PT-BB), an optimization algorithm exploiting pseudo-tree arrangements. We show that general ideas developed for the CSP case apply directly to soft constraint problems. However, a naive implementation is not competitive with state-of-the-art algorithms. The reason is that independence in optimization subproblems means *solution independence* (i.e., the global solution can be correctly computed by solving subproblems separately). However, there is not *efficiency independence* because solving subproblems separately produces loose upper bounds for each subproblem. Loose upper bounds imply weak pruning capabilities and, consequently, an inefficient execution. We overcome this problem with the use of *local* upper bounds, which may improve over a loose global upper bound.

2 PRELIMINARIES

A binary *weighted constraint satisfaction problem* (WCSP) is a triple $P = (\mathcal{X}, \mathcal{D}, \mathcal{C})$. $\mathcal{X} = \{1, \dots, n\}$ is a set of *variables*. Each variable $i \in \mathcal{X}$ has a finite *domain* $D_i \in \mathcal{D}$ of values that can be assigned to it. (i, a) denotes the assignment of value $a \in D_i$ to variable i. A tuple t is an ordered set of values assigned to the ordered set of variables $\mathcal{X}_t \subseteq \mathcal{X}$. If B is a subset of $\mathcal{X}_t$, the projection of t over B is noted as $t\downarrow_B$. $\mathcal{C}$ is a set of binary soft constraints. A soft constraint C_{ij} is a cost function over binary tuples (i.e., $C_{ij} : D_i \times D_j \to \mathbb{N}$). The *cost* of tuple t, noted $cost(t)$, is the sum of all applicable costs,

$$cost(t) = \sum_{C_{ij} \in \mathcal{C},\ \{i,j\} \subseteq X_t} C_{ij}(t\downarrow_{\{i,j\}})$$

The goal is to *find a complete assignment with minimum valuation*,

$$\min_{t\ s.t.\ \mathcal{X}_t = \mathcal{X}} \{cost(t)\}$$

Branch-and-bound (BB) is a *search* algorithm for WCSP solving [7, 5, 9]. It traverses the search tree defined by the problem, where internal nodes represent incomplete assignments and leaf nodes stand for complete ones. During the traversal, BB keeps the cost of the best solution found so far. Its cost is an *upper bound* (ub) of the best cost in the problem. At each internal node, defined by its current partial assignment t, the algorithm computes a *lower bound* (lb) which underestimates the best solution that can be found by extending t. When $ub \leq lb$, the current best cost cannot be improved by extending t. Consequently, the algorithm *backtracks* pruning the subtree below the current node t. The time complexity of BB is exponential on the number of variables and the space complexity is polynomial. At a given node, $ub - lb$ is what we call the *uncertainty gap*. It is well

[1] Dep. LSI, UPC, Jordi Girona 1-3, 08034 Barcelona, Spain
[2] IIIA-CSIC, Campus UAB, 08193 Bellaterra, Spain
[3] IIIA-CSIC, Campus UAB, 08193 Bellaterra, Spain

```
function BB(t, F, D, C, ub) return nat;
1 if (F = ∅) then return min{ub, cost(t)};
2 else
3   i ← PopVar(F);
4   for each a ∈ D_i do
5     newt ← t + (i, a);
6     lb ← LB(newt, F, D, C);
7     if (lb < ub) then
8       D' ← LookAhead(newt, F, D, C, ub);
9       if (¬EmptyDom(D')) then lb ←BB(newt, F, D', C, ub);
10    if (lb < ub) then ub ← lb;
11  return ub;
endfunction
```

Figure 1. Branch and Bound.

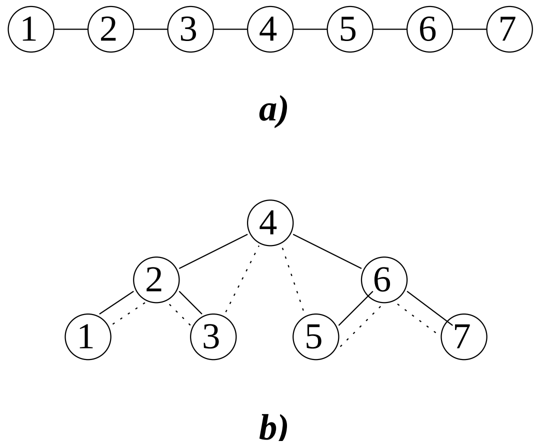


Figure 2. a) A constraint graph and b) a pseudo-tree arrangement.

known [5, 11, 6] that the average efficiency of BB heavily depends on the availability of good bounds producing small uncertainty gaps at initial levels of the search tree.

Figure 1 shows BB, a recursive branch-and-bound based algorithm enhanced with a look-ahead process in which unfeasible values are pruned [5]. BB receives the current problem defined by the tuple containing the current assignment t, the set of *future* (unassigned) variables $\mathcal{F}$, the current domains $\mathcal{D}$, the set of constraints $\mathcal{C}$ and the global upper bound ub. If the cost of the best extension of t is less than ub, the algorithm returns that cost. Else, the algorithm returns ub. Therefore, the behavior of BB is defined as,

$$\text{BB}(t, \mathcal{F}, \mathcal{D}, \mathcal{C}, ub) = \min\{ub, \min_{t'}\{cost(t')\}\}$$

where t' is an extension of t to variables in $\mathcal{F}$.

BB works as follows: If the set $\mathcal{F}$ is empty, the result is trivially computed (line 1). Else, it selects a variable i and iterates over its values (lines 3, 4). For each value $a \in D_i$ the current assignment t is extended to (i, a) and stored in $newt$ (line 5). Next, the algorithm computes a lower bound lb (line 6). If the lower bound is greater than or equal to ub, the current subproblem does not need to be solved because there is no solution improving over ub (line 7). Therefore, the algorithm proceeds to the following domain value. Else, a look-ahead procedure is executed in which unfeasible values are removed from future domains (line 8). If no empty domain is detected, the current problem is recursively solved with ub as global upper bound, and the solution is stored in variable lb (line 9). If lb is smaller than the global upper bound ub, a better solution has been found so ub is updated (line 10). After trying all feasible values of variable i, the cost of the best solution remains in ub, which is returned (line 11).

3 PSEUDO-TREE SEARCH

The *constraint graph* of a CSP instance is an undirected graph having problem variables as nodes and edges connecting pairs of constrained variables. A *pseudo-tree arrangement* of a constrained graph [4, 1] is a rooted tree with the same set of vertices as the constraint graph and the property that adjacent vertices from the constraint graph must be in the same branch of the rooted tree. Figure 2.a shows a constraint graph of a CSP with seven variables and six constraints. For each variable $i = 1..6$, there is a constraint $C_{i,i+1}$. Figure 2.b shows one of the many pseudo-tree arrangements that are possible. Solid lines indicate the edges in the pseudo-tree. Dotted lines are the edges in the original constraint graph.

Pseudo-tree search for CSPs [4] assigns variables according to a pseudo-tree arrangement. Starting from the pseudo-tree root, if the current variable has $q > 1$ children in the pseudo-tree, the current problem can be divided into q independent subproblems. Each subproblem includes previous assignments in the path from the subproblem to the root. These subproblems can be solved independently. The current problem has a solution iff every independent subproblem has a solution. Pseudo-tree search has time complexity $O(l\ exp(h))$, where l and h are the number of leaves and the height of the pseudo-tree, respectively. Pseudo-tree search is polynomial in space.

The extension of pseudo-tree search to WCSP solving requires the introduction of the previous ideas to the branch-and-bound scheme. Let us illustrate it through an example: Consider a WCSP instance $(\mathcal{X}, \mathcal{D}, \mathcal{C})$, whose variables can be partitioned into three sets $\mathcal{X} = \mathcal{W} \cup \mathcal{Y} \cup \mathcal{Z}$ such that no constraint connects variables from $\mathcal{Y}$ and $\mathcal{Z}$. Assume that we are solving the problem with algorithm BB. The current upper bound is ub, the current assignment is t, which assigns all variables in $\mathcal{W}$. Therefore, the current problem is $P = (t, \mathcal{F}, \mathcal{D}, \mathcal{C})$, where the set of future variables is $\mathcal{F} = \mathcal{Y} \cup \mathcal{Z}$. P is separable into two subproblems: $P_Y = (t, \mathcal{Y}, \mathcal{D}_Y, \mathcal{C}_Y)$ and $P_Z = (t, \mathcal{Z}, \mathcal{D}_Z, \mathcal{C}_Z)$. $\mathcal{D}_Y$ and $\mathcal{D}_Z$ denote the current domains of variables in $\mathcal{Y}$ and $\mathcal{Z}$, respectively. $\mathcal{C}_Y$ and $\mathcal{C}_Z$ denote the set of constraints mentioning variables in $\mathcal{Y}$ and $\mathcal{Z}$, respectively. These two problems are independent because they do not share any constraint.

Let v_Y and v_Z be the cost of the optimal solution to P_Y and P_Z, respectively. Clearly, the optimal cost of P (that is, the minimum cost among assignments including tuple t) can be computed as $v = v_Y + v_Z - cost(t)$, since P_Y and P_Z have no constraints in common. Note that $cost(t)$ needs to be substracted, because it has been counted twice. Therefore, P can be solved by solving its independent subproblems P_Y and P_Z separately.

Consider now that we want to solve P_Y and P_Z by two independent calls to BB, starting with P_Y. We must call BB with P_Y and an appropriate upper bound ub_Y. Aiming at efficiency, we want ub_Y as low as possible, in order to decrease the uncertainty gap for the subprobem. The simplest idea is to use $ub_Y = ub$, the maximum acceptable cost for the whole problem P. A better approach is to compute lb_Z, a lower bound of the cost of solving P_Z. Then, $lb_Z - cost(t)$ is a necessary cost of extending t to $\mathcal{Z}$. Consequently, we can use

$ub_Y = ub - lb_Z + cost(t)$, because any higher cost in P_Y cannot be possibly extended to $\mathcal{Z}$ with cost below ub. This approach may still be too weak when lb_Z is a bad lower bound, because we are passing to the *local* task of solving P_Y the *global* uncertainty gap in P. One way to overcome this problem, is to compute a local upper bound lub_Y of the cost solution of P_Y. If $ub - lb_Z + cost(t)$ is a bad upper bound, it may not be costly to find a lub_Y below $ub - lb_Z + cost(t)$. Then, we can use $ub_Y = \min\{lub_Y, ub - lb_Z + cost(t)\}$ when solving P_Y.

After solving P_Y, we must call BB with P_Z and an appropriate upper bound ub_Z. We can set ub_Z to $ub - v_Y + cost(t)$, the cost that we have left for P_Z after solving P_Y. Again, we can compute a local upper bound lub_Z and take the minimum between the two upper bounds, but now that we have the actual solution of P_Y, it is more unlikely that we can improve over ub_Z.

Figure 3 shows PT-BB, which implements pseudo-tree branch and bound. It extends the ideas discussed in the previous example to an arbitrary number of independent subproblems. PT-BB assumes that variables are selected according to a pseudo-tree arrangement.

The following notation is used: $(t, \mathcal{F}, \mathcal{D}, \mathcal{C})$ is the problem with which the procedure is called and ub is the global upper bound. If the set $\mathcal{F}$ is empty, the result is trivially computed (line 1). Else, it selects a variable i and iterates over its values (lines 3, 4). Each value a defines the current problem $P = (newt, \mathcal{F}, \mathcal{D}, \mathcal{C})$, which is decomposed into a set of q independent subproblems, $P_k = (newt, \mathcal{F}_k, \mathcal{D}_k, \mathcal{C}_k)$, with $k = 1..q$, $q > 0$, one per child of i in the pseudo-tree arrangement. For each P_k, a lower bound lb_k is computed (line 6). The lower bound of P is lb, computed as the sum of independent lower bounds, removing all contributions but one of $cost(newt)$ (line 7),

$$lb = (\sum_{k=1}^{q} lb_k) - (q-1)\ cost(newt)$$

Independent subproblems are sequentially solved (line 8). The global upper bound of P is ub. Using ub and the independent subproblem lower bounds, the algorithm specializes the global upper bound to each independent subproblem as (line 9),

$$ub_k = ub - lb + lb_k$$

In addition, local upper bound lub_k are computed (line 10). The upper bound for each independent subproblem is the minimum between the two available bounds (line 11),

$$newUb_k = \min\{ub_k, lub_k\}$$

If the lower bound lb_k of the subproblem is greater than or equal to $newUb_k$, the current subproblem does not need to be solved because either lub_k is the solution, or there is no solution improving over ub (line 12). Therefore, the algorithm proceeds to the following subproblem. Otherwise, a look-ahead procedure is executed in which unfeasible values are removed from future domains (line 13). If no empty domain is detected, the current problem is recursively solved with $newUb_k$ as global upper bound, and the solution is stored in variable lb_k (line 15). As the algorithm solves independent subproblems, the lower bound lb of P is improved by replacing the lower bounds by the actual solution (line 16). If during the process, the lower bound lb becomes greater than or equal to the global upper bound ub, search can be aborted reporting that P cannot be solved with a cost less than ub (line 17). Once all independent subproblems have been solved, if lb is smaller than the global upper bound ub, a better solution has been found so ub is updated (line 18). After trying all feasible values of variable i, the cost of the best solution remains in ub, which is returned (line 19).

function PT-BB$(t, \mathcal{F}, \mathcal{D}, \mathcal{C}, ub)$ **return** nat;
1 **if** $(\mathcal{F} = \emptyset)$ **then return** $\min\{ub, cost(t)\}$;
2 **else**
3 $i \leftarrow$ PopVar$(\mathcal{F})$;
4 **for each** $a \in D_i$ **do**
5 $newt \leftarrow t + (i, a)$;
6 **for each** $k = 1..q$ **do** $lb_k \leftarrow$ LB$(newt, \mathcal{F}_k, \mathcal{D}_k, \mathcal{C}_k)$;
7 $lb \leftarrow (\sum_{k=1}^{q} lb_k) - (q-1)\ cost(newt)$;
8 **for each** $k = 1..q$ **do**
9 **if** $(lb \geq ub)$ **then exit for**;
10 $ub_k \leftarrow ub - lb + lb_k$;
11 $lub_k \leftarrow$ UB$(newt, \mathcal{F}_k, \mathcal{D}_k, \mathcal{C}_k)$;
12 $newUb_k \leftarrow \min\{ub_k, lub_k\}$;
13 **if** $(lb_k < newUb_k)$ **then**
14 $\mathcal{D}'_k \leftarrow$ LookAhead$(newt, \mathcal{F}_k, \mathcal{D}_k, \mathcal{C}_k, newUb_k)$;
15 **if** $(\neg$EmptyDom$(\mathcal{D}'_k)$) **then**
16 $lb_k \leftarrow$PT-BB$(newt, \mathcal{F}_k, \mathcal{D}'_k, \mathcal{C}_k, newUb_k)$;
17 $lb \leftarrow (\sum_{k=1}^{q} lb_k) - (q-1)\ cost(newt)$;
18 **if** $(lb < ub)$ **then** $ub \leftarrow lb$;
19 **return** ub;
endfunction

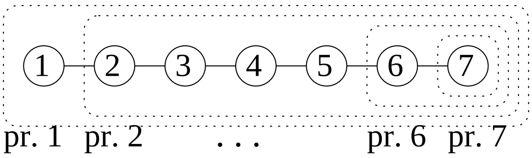
Figure 3. Pseudo-Tree Branch-and-Bound.

4 COMBINING PSEUDO-TREE AND RUSSIAN DOLL SEARCH

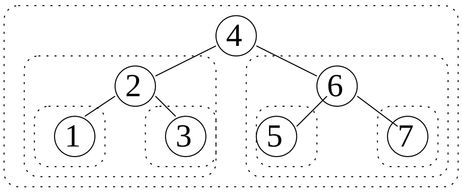
Russian Doll Search (RDS) [11, 8] is a BB algorithm which invests in high quality lower bounds. The idea of Russian Doll is to replace one search by n successive searches on nested subproblems. Given an ordering o of the problem variables, subproblem i involves all the variables from the ith variable to the last, and subproblem 1 is the whole problem. Figure 4.a depicts RDS nested problems for the constraint graph of Figure 2.a along the lexicographic variable ordering. In RDS, subproblems are solved sequentially in inverse order, starting with subproblem n (here, we use a subproblem notation that is inverse to the one appearing in [11]). All subproblems use the same static variable ordering o restricted to the variables of the subproblem. After solving the subproblem i, RDS records the best solution $Sol(i)$ along with its cost $Cost(i)$. The key to the efficiency of this method lies in that $Cost(i)$ can be used in local lower bounds of subsequent problems, at nodes where the set of future variables is exactly $\{i, i+1, \ldots, n\}$. An additional feature of RDS is that the cost of the assignment that extends $Sol(i)$ to any value of variable $i-1$ produces an upper bound of the cost of subproblem $i-1$. Therefore, RDS provides, at almost no extra cost, local upper bounds[4].

RDS nicely adapts to PT-BB search. The idea is to use the pseudo-tree arrangement in the nested subproblems structure. Given a pseudo-tree arrangement, the subproblem i involves all the variables in the sub-tree rooted by node i. If node i is the pseudo-tree root, subproblem i is the whole problem. Figure 4.b depicts the nested problems for the pseudo-tree arrangement of Figure 2.b. Each subproblem is solved after each of its children subproblems have

[4] A different issue is the quality of this upper bound, which will depend on how close this extension is with respect to $Sol(i-1)$.



a)



b)

Figure 4. The nested structure of $a)$ RDS and $b)$ PT-RDS.

been solved. After solving subproblem i, we record its best solution $Sol(i)$ along with its cost $Cost(i)$. We call this algorithm pseudo-tree RDS (PT-RDS). When solving the whole problem, variables are assigned following the pseudo-tree arrangement. When arriving to subproblem i, a local upper bound can be computed as the cost of the assignment that extends $Sol(i)$ with the assigned variables that are in the path from i to the root. Therefore, PT-RDS provides local upper bounds of subproblems considered by pseudo-tree search.

5 EXPERIMENTAL RESULTS

We have evaluated the performance of pseudo-tree RDS using the SRDS algorithm (the specialized version of RDS [8]), on over-constrained binary random CSP. A binary random CSP class is characterized by $\langle n, d, p_1, p_2 \rangle$ where n is the number of variables, d the number of values per variable, p_1 the graph *connectivity* defined as the ratio of existing constraints, and p_2 the constraint *tightness* defined as the ratio of forbidden value pairs. The constrained variables and the forbidden value pairs are randomly selected [10]. Using this model, we have tested on the connectivity range between 0.1 and 0.5, where non-degenerated pseudo-trees can be constructed. Specifically, we have experimented on the following problem classes,

1 $\langle 15, 10, 50/105, p_2 \rangle$, 2 $\langle 20, 5, 100/190, p_2 \rangle$,
3 $\langle 14, 7, 27/91, p_2 \rangle$, 4 $\langle 20, 5, 57/190, p_2 \rangle$,
5 $\langle 25, 10, 37/300, p_2 \rangle$, 6 $\langle 40, 5, 55/780, p_2 \rangle$.

Observe that (1) and (2) are medium connected problems ($p_1 = 0.5$), (3) and (4) are problems with low connectivity ($p_1 = 0.3$), and (5) and (6) are sparse problems ($p_1 = 0.1$). For each problem class and each parameter setting, we generated samples of 50 instances.

Each problem is solved by three algorithms based on PFC [5]: MRDAC, SRDS and PT-SRDS. In MRDAC variables are dynamically ordered by the increasing ratio of domain size divided by forward degree, and values are ordered by increasing $ic + dac$ (see [6] for a more detailed description). SRDS uses a static variable ordering that heuristically combines degree and locality, to produce low bandwidth orderings. PT-SRDS follows the variable ordering existing in the pseudo-tree. Since RDS-based algorithms are very sensitive to variable orderings, we decide to construct the pseudo-tree according to the heuristic static variable ordering used for SRDS. This means that the ordering in which variables appear in the pseudo-tree branches is in agreement with that static variable ordering.

Figure 5 reports the average cpu-time required to solve the six problem classes. We observe that for medium connectivity classes ($p_1 = 0.5$), SRDS and PT-SRDS both offer the best performance (without practical differences) while MRDAC is clearly worse. For low connectivity classes ($p_1 = 0.3$), PT-SRDS shows the best performance, while there is no clear second between SRDS and MRDAC. For sparse problems ($p_1 = 0.1$), PT-SRDS exhibits the best performance, followed by MRDAC and SRDS in third position. From these graphs we conclude that, for random binary problems with medium to very low connectivity, PT-SRDS is the algorithm of choice. PT-SRDS performance improves over other algorithms as connectivity decreases. This observation is in full agreement with our approach, because a low connectivity problem is very likely to produce a shallow pseudo-tree, which enhances the savings of pseudo-tree search with respect to other algorithms.

Figure 6 contains the average number of visited nodes for three problem classes (the other results are omitted for space reasons). We observe that PT-SRDS visits fewer nodes than SRDS for the six problem classes tested. This suggest that PT-SRDS searches more efficiently than SRDS. However, PT-SRDS has a higher overhead than SRDS. The savings of pseudo-tree search are compensated by this extra overhead in medium connectivity classes, both algorithms offering a similar performance. For low and very low connectivity, pseudo-tree savings surpass the pseudo-tree overhead.

Since the temporal complexity is exponential in the pseudo-tree height, we have recorded the average height of pseudo-trees used in the experiments. For medium connectivity classes, the average height is around $0.6n$ (n is the number of variables); for low connectivity classes is around $0.5n$; for very low connectivity classes is $0.3n$. These results confirm that pseudo-tree search decreases substantially the height of the search tree to explore, improving efficiency.

We have explored the relevance of local upper bounds in our implementation, substituting line 10 of Figure 3 by $lub_k \leftarrow \infty$. In this case, experimental results show that PT-SRDS looses performance. This result confirms that, although solving independent subproblems independently is an attractive strategy, working with the global upper bound is not cost-effective and the presence of local upper bounds is essential.

If we eliminate from the problem the constraints that are less tight, better pseudo-trees can be build (with lower height). Solving the problem with this particular pseudo-tree will give us a lower bound on the optimum cost of the original problem. Doing so we have obtained good lower bounds in few seconds for radio link frequency assignment (CELAR) instances [2].

6 CONCLUSION

Pseudo-tree search is a well known algorithm for CSP with two nice properties: $i)$ its time complexity is bounded by a structural parameter and $ii)$ its space complexity is polynomial. In this paper, we have extended pseudo-tree search to the soft constraints framework. We have shown that the general principles can be easily extended. However, if good average efficiency is required, a careful implementation is needed. We have introduced PT-BB, a branch-and-bound algorithm that performs pseudo-tree search. Its main feature is that it computes local upper bounds with which a good efficiency is obtained. We have also shown that pseudo-tree search nicely combines

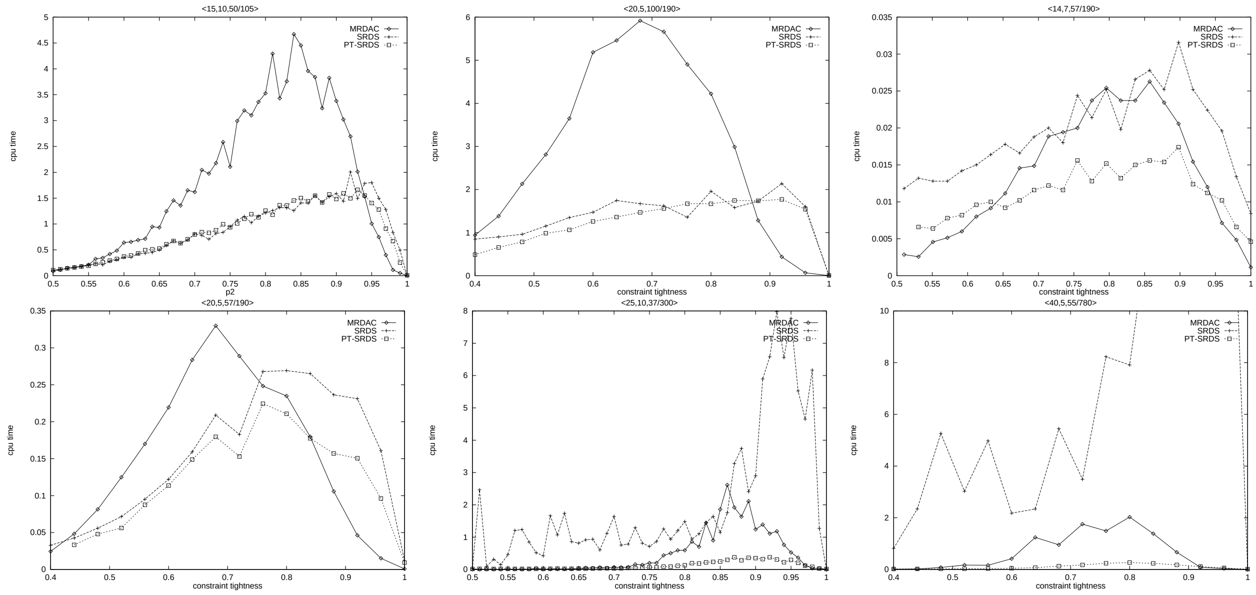


Figure 5. Average CPU time versus tightness for six classes of binary random problems.

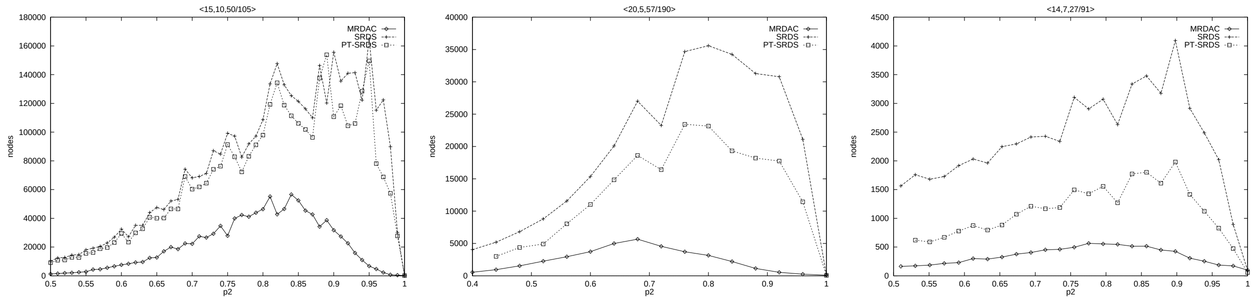


Figure 6. Average visited nodes versus tightness for three classes of binary random problems.

with russian doll search, producing an efficient algorithm.

We have presented some preliminary empirical results showing how our algorithms are competitive with state-of-the-art solvers. We believe that our results can be further improved, because we have still not addressed several important practical aspects. For instance, we have not studied the effect of the ordering in which the independent subproblems are solved. Similarly, we have used naive pseudo-tree arrangements, probably having non-optimal height. The effect of this on the algorithms efficiency is a topic of our current work.

ACKNOWLEDGEMENTS

This work was supported by the IST Programme of the Commission of the European Union through the ECSPLAIN project (IST-1999-11969), and by the Spanish CICYT project TAP99-1086-C03. We thank the anonymous reviewers for their constructive criticisms.

REFERENCES

[1] R. Bayardo and D. Miranker, 'On the space-time trade-off in solving constraint satisfaction problems', in *Proc. of 14th IJCAI*, pp. 558–562, (1995).

[2] B.Cabon, S. De Givry, L.Lobjois, T.Schiex, and J.P.Warners, 'Radio link frequency assignment', *Constraints*, **4(1)**, 79–89, (1999).

[3] S. Bistarelli, U. Montanari, and F. Rossi, 'Semiring-based constraint satisfaction and optimization', *Journal of the ACM*, **44**(2), 201–236, (1997).

[4] E.C. Freuder and M.J. Quinn, 'Taking advantage of stable sets of variables in constraint satisfaction problems', in *Proc. of 9th IJCAI*, pp. 1076–1078, (1985).

[5] E.C. Freuder and R.J. Wallace, 'Partial constraint satisfaction', *Artificial Intelligence*, **58**, 21–70, (1992).

[6] J. Larrosa, P. Meseguer, and T. Schiex, 'Maintaining reversible DAC for max-CSP', *Artificial Intelligence*, **107**(1), 149–163, (1999).

[7] E. L. Lawler and D. E. Wood, 'Branch-and-bound methods: A survey', *Operations Research*, **14(4)**, 699–719, (1966).

[8] P. Meseguer and M. Sanchez, 'Specializing russian doll search', in *Proc. of 7th CP*, pp. 464–478, (2001).

[9] T. Schiex, H. Fargier, and G. Verfaillie, 'Valued constraint satisfaction problems: hard and easy problems', in *Proc. of 14th IJCAI*, pp. 631–637, (1995).

[10] B. Smith, 'Phase transition and the mushy region in constraint satisfaction problems', in *Proc. of 11th ECAI*, pp. 100–104, (1994).

[11] G. Verfaillie, M. Lemaître, and T. Schiex, 'Russian doll search', in *Proc. of 13th AAAI*, pp. 181–187, (1996).