

# Merging constraint satisfaction subproblems to avoid redundant search\*

Javier Larrosa

Dep. Llenguatges i Sistemes Informàtics

Universitat Politècnica de Catalunya

Jordi Girona Salgado, 1-3

E-08034 Barcelona, Spain

## Abstract

When solving a *constraint satisfaction problem* we may find similar subproblems. Algorithms not exploiting this similarity are condemned to duplicate some work. In this paper we introduce a technique for merging sibling subproblems which avoids redundancy in search associated to their similarity. We show that, when *forward checking* is enhanced with this capability, its performance may be increased. Experimental results on hard crossword puzzles support the practical validity of our approach.

## 1 Introduction

*Constraint Satisfaction Problems* (CSPs) involve finding values for problem variables subject to constraints on which combinations of values are permitted. They have many applications in *artificial intelligence* and other areas of *computer science*.

It is believed that some real CSPs have different values for the same variable that behave in a similar manner [Freuder, 1991; Haselbock, 1993]. *Depth-first* algorithms (such as *forward checking* [Haralick and Elliott, 1980]) not exploiting this similarity are condemned to do some redundant work. Figure 1a (left upper corner) shows a simple colouring graph problem used to illustrate this idea. The problem is to assign a colour to each node in such a way that adjacent nodes have different colours. Colour choices for each node are represented by letters and included in the nodes. Figure 1b shows the search tree traversed by forward checking before finding that the problem is unsolvable (variables and values are selected lexicographically).

It can be observed that the first two subtrees (after assigning A and B to  $x_1$ ) have a large similar area (shadowed). The reason for this is that values A and B behave in a very similar manner. Thus, their assignment produces similar subproblems. This fact can be observed in Figure 1a, which also shows the three subproblems obtained

after assigning the three alternative colours to  $x_1$  and updating the domains. As can be seen, the first and the second have more in common than the third.

We claim that solving similar subproblems without exploiting their similarities is inefficient since an important part of the work is, somehow, duplicated. Obviously, it is desirable to avoid the unnecessary repetition of as much work as possible. This paper aims in that direction. We show that, if sibling subproblems are close enough, it is worth to *merge* their subtrees, and only distinguish to which value they are committed at deeper levels of the tree.

We introduce a technique for merging sibling subproblems based in the concept of *weak assignment*. Briefly, a weak assignment is the reduction of a variable

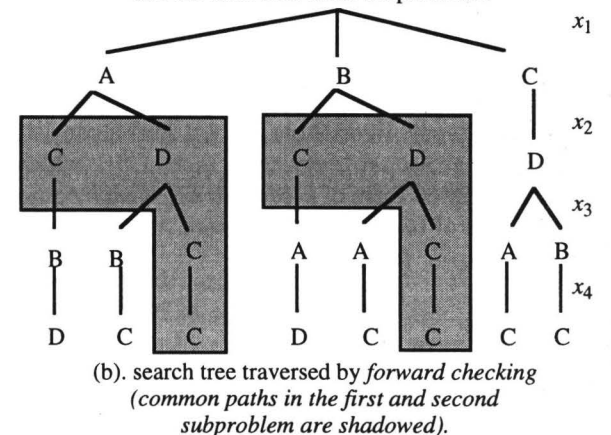
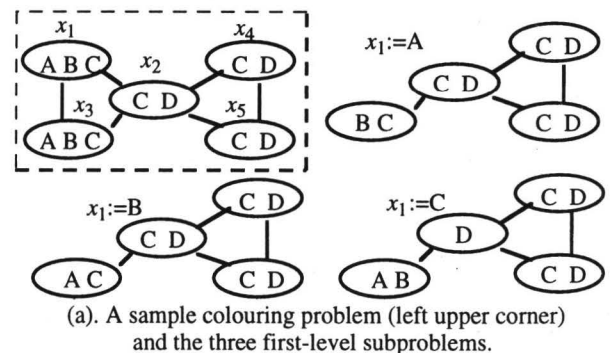


Figure 1. A colouring graph and the search tree traversed by forward checking.

\*This research is supported by the Spanish CICYT project TIC96-0721-C02-02.

domain to a subset of its feasible values. The resulting subproblem merges into one the set of subproblems that would be obtained if the subset of values had been assigned in the classical way. If one delays the decision of what value is definitively assigned to the variable, the resulting search tree is the fusion of the set of trees. We show that this technique is especially convenient to merge similar subproblems.

We have introduced weak assignments into forward checking and we show that, for specific domains, its performance is significantly increased. Our experimental results on hard random crossword puzzles support the practical validity of our approach.

This paper is structured as follows. In Section 2 we introduce the notation and definitions that will be used throughout the paper. In Section 3 we present the concept of weak assignment and analyze its effectiveness for merging subtrees. In Section 4 we introduce a modification to forward checking enhancing it with merging capabilities. In Section 5 we report our experiments on crossword puzzles. In Section 6 we review and compare work related to ours. Finally, in Section 7 we give the conclusions of our work.

## 2 Definitions and Notation

A discrete binary CSP is defined by a set of  $n$  variables  $\{x_i\}$  taking values on discrete and finite domains  $\{d_i\}$  under a set of binary constraints  $\{R_{ij}\}$  ( $i, j = 1, \dots, n$ ). A constraint  $R_{ij}$  is a subset of  $d_i \times d_j$ , containing the permitted values for  $x_i$  and  $x_j$ . A solution to the CSP is an assignment of values to variables satisfying every constraint. This paper is devoted to the task of finding one or all solutions to a CSP.

Given a CSP, a subproblem is what remains after the assignment of some of its variables. Assigned variables are called *past* variables and denoted by  $P$ . Unassigned variables are called *future* variables and denoted by  $F$ . A *subproblem*  $S$  is defined by a tuple  $S = \langle P, F, D \rangle$  where  $P = \{x_i = k_i\} \forall i \in P$  are the assignments to past variables,  $F$  is the set of future variables and  $D = \{d_j\} \forall j \in F$  are the non empty sets of future values which remain feasible.

*Forward checking* (FC) is a simple, yet powerful algorithm for constraint satisfaction. Given a subproblem  $S$ , FC selects an unassigned variable  $x_i$  and transforms  $S$  into a set of simpler subproblems  $\{S^l \mid l \in d_i\}$ . Each subproblem  $S^l = \langle P^l, F^l, D^l \rangle$  is defined by  $P^l = P \cup \{x_i = l\}$ ,  $F^l = F - \{x_i\}$ ,  $D^l = \{d_j^l\} \forall j \in F^l$ . Domain  $d_j^l$  is the updating of  $d_j$  after removing those values unfeasible with the new assignment  $x_i = l$ . Variable  $x_i$  is the *current variable* and it changes from being a future variable in  $S$  to a past variable in  $S^l$ .

FC can be seen as the depth-first traversal of a search tree where each subproblem is a tree node. Two nodes are *siblings* if they have the same parent. If all solutions are required, the whole tree is traversed. If one solution is wanted, the algorithm stops when it finds one. We denote by  $T$  the subtree rooted by node  $S$ . An important parameter related to the computational effort needed to solve a problem is the number of nodes that the algorithm requires to visit when traverses the whole search tree (ITI). It is especially appropriate when: (i) all solutions are required, or (ii) only one solution is wanted, but the problem is not

trivial and a significant part of the search tree has to be traversed before finding a solution.

To evaluate the validity of our approach we suppose that forward checking uses a *static variable ordering* (the current variable selection depends on the tree level where it takes place). However, our approach is still valid when *dynamic variable orderings* are used, as will be shown in section 5.

Throughout this paper we will refer to a generic subproblem,  $S$ , and to its successors  $\{S^l \mid l \in d_i\}$  (being  $x_i$  the current variable). Specific successors will be denoted by  $S^a, S^b, \dots$ , where their index corresponds to the value assigned to  $x_i$ . To differentiate children elements from their parent we will use their index (*i.e.*:  $S^a = \langle P^a, F^a, D^a \rangle$ ,  $T^a$ , etc.).

**Definition 1:** Given  $S, S^a, S^b$ , and two of its descendants  $S^a \in T^a$  and  $S^b \in T^b$ , we say that  $S^a$  and  $S^b$  follow common paths after  $S$  if their assignments to past variables ( $P^a$  and  $P^b$ ) only differ in the assignment made to  $x_i$ .

Therefore, common paths are those paths only differing in the assignment that caused their separation (for instance, in Fig. 1b (A,D,C,C) and (B,D,C,C) are common paths). Since we are assuming a static variable ordering, proceeding upwards from  $S^a$  and  $S^b$  level by level, one encounters pairs of nodes which follow common paths, too.

Finally, *similarity* among subproblems is a graded relation which in this work is restricted to sibling nodes. For our purposes it is enough to give an intuitive definition. Similarity must measure how close are future variable domains. Thus, two subproblems have the lowest similarity if they have no common values in their domains. Two subproblems have the highest similarity if they have exactly the same values in their domains.

## 3 Merging Subproblems

### 3.1 Weak Assignments

Common paths between two subproblems define two subtree areas with identical arboreous structure (shadowed subtree areas in Fig. 1b correspond to common paths between the first and second subproblem). Each subtree node has its corresponding node at the other subtree which only differs in: (i) one assignment and (ii) the effect of that assignment on future variable domains. Common paths are, somehow, redundant since, except for one assignment, their nodes represent the same situation. It would be desirable to generate them only once and employ them for both subproblems. In what follows, we introduce an approach to this idea based on the concept of *weak assignment*.

**Definition 2:** Given a subproblem  $S = \langle P, F, D \rangle$ , a variable  $x_i$ , and a subset  $K \subset d_i$  such that  $\text{card}(K) > 1$ , the weak assignment  $x_i := K$  produces a new simpler subproblem  $S^K = \langle P, F, D^K \rangle$  where

- (i)  $d_i^K = K$
- (ii)  $d_j^K = \{a \in d_j \mid a \text{ consistent to at least one value in } K\}$

In words, a weak assignment reduces the domain of an unassigned variable and prunes future values that are

The set of solutions of  $S^K$  is equal to the set of solutions of  $\{S^l \mid l \in K\}$ . Therefore, one can either solve  $S^K$  or the set  $\{S^l \mid l \in K\}$ . Solving  $S^K$ , one can delay the decision of what value (among  $K$ ) is assigned to  $x_i$  to deeper levels of the tree (unless otherwise indicated, we will assume that, after a weak assignment, the variable remains unassigned until the last tree level). The advantage of solving  $S^K$  is that all common paths in  $\{T^l \mid l \in K\}$  occurring before the last tree level are *merged* into a unique path, which considers, simultaneously, the different assignments to  $x_i$ . However, look ahead after a weak assignment has a lower pruning power. Therefore, the detection of domain wipe-outs may be delayed.

### 3.2 Effectiveness of Merging Subproblems

To evaluate the effectiveness of merging two subproblems  $S^a$  and  $S^b$  with a weak assignment  $x_i = \{a, b\}$  we compare the size of the subtree underneath  $S^{[a,b]}$  (i.e.:  $|T^{ab}|$ ) with the size of the subtrees under  $S^a$  and  $S^b$  ( $|T^a| + |T^b|$ ). As a first step, we partition the set of descendant

$$|T^a| + |T^b| = |V^a| + |W^a| + |Z^a| + |V^b| + |W^b| + |Z^b|$$
$$|T^a|+|T^b|=|V^a|+|Z^a|+|V^b|+|Z^b|+2|W^a|$$
**Observation 2:**

- These observations establish a one-to-one mapping between some nodes in  $T^{ab}$  and nodes in  $T^a$  and  $T^b$ . An important consequence is that any node in  $T^a$  or  $T^b$  has its associated node in  $T^{ab}$ . However, observation (iii) shows that non solution nodes in the common paths of  $T^a$  and  $T^b$  share their associated node in  $T^{ab}$  (this is why the fusion occurs).

**Proof:** Consider a node  $S^r \in T^{ab}$ , whose assignments to past variables include two assignments,  $x_p := l_p$  and  $x_q := l_q$  such that  $l_p \in d_p^a - d_p^b$  and  $l_q \in d_q^b - d_q^a$ . This node is not mapped to  $T^a$ , nor to  $T^b$ .

$$|T^{ab}| \geq |V^a| + |V^b| + |Z^a| + |Z^b| + |W^a|$$

If  $S^a$  and  $S^b$  are very similar, it is natural to expect  $T^a$  and  $T^b$  to have many common paths. It is so, because the only chance that  $S^a$ , descendent of  $S^a$ , does not have its common node  $S^b$ , descendent of  $S^b$ , is due to the dissimilarities between  $S^a$  and  $S^b$ . It may happen for two reasons: (i)  $S^a$  has a value assigned to a variable which is unfeasible in  $S^b$ , or (ii) assignments made to  $S^a$  would produce a domain wipe-out if made at  $S^b$ .

### 3.3 Nested Weak Assignments

Tree size reductions obtainable by merging subproblems can be increased if more than one weak assignment is performed through a tree path (*nested* weak assignments). Thus, if a weak assignment  $x_2 := \{c, d\}$  is performed after a weak assignment  $x_1 := \{a, b\}$ , descendant nodes merge those subproblems considering the four possible combination for  $x_1$  and  $x_2$ . In general,  $u$  nested weak assignments of  $v$  values each, can merge  $v^u$  nodes differing in  $u$  assignments.

However, the pruning power of the look ahead decreases with nested weak assignments because weaker information for pruning is available. Therefore, the detection of domain wipe-outs may be delayed to deeper nodes, producing the addition of new nodes. The benefit of nested weak assignments depends on the balance of these two factors.

### 4 FC with Weak Assignments

In this section we modify forward checking adding to it the capability of merging subproblems with nested weak assignments. The resulting algorithm,  $FC_w$ , is outlined in Figure 3. At each node,  $FC_w$  selects a variable  $x_i$  and partitions its domain into groups. Groups with more than one element are weakly assigned producing a new subproblem where  $x_i$  remains being a future variable, groups with a single value are assigned in the classical way producing a new subproblem where  $x_i$  becomes a past variable (therefore, if the domain is partitioned into groups of one value each, the algorithm behaves as FC). The look-ahead function removes from future variable domains those values inconsistent with *all* values in the group under consideration (therefore, if the group has only one value, this procedure behaves as the standard look-ahead).

The *select\_var* function implements the variable ordering heuristic of choice, which can be either static or dynamic. However, some criterion must be added to delay the definitive assignment of variables that are weakly assigned. In our implementation, when a variable is selected and weakly assigned, it is not selected again until the rest of variables have been selected.

The *domain\_partition* function is a key element for the algorithm efficiency. This function must group values producing sufficiently similar subproblems. The implementation of this function raises two important questions: (i) how to measure subproblem similarity, and (ii) what is the similarity threshold to cluster values after which  $FC_w$  is cost effective. We believe that the answer to these questions is domain dependent (as well as the existence of similar subproblems). For instance, in *random problems* (a typical benchmark for CSP algorithms) weak assignments seem not to be useful because their random nature makes improbable the occurrence of very similar subproblems. Moreover, they do not have any structure, therefore no domain knowledge can be used to evaluate similarity. On the other hand, more structured problems may be more appropriate for our approach because some domain specific knowledge can be used to detect subproblem similarities. We will come back to this point in the next section.

```

Procedure  $FC_w$ (subproblem  $\langle P, F, D \rangle$ )
if (empty_domain( $D$ )) then backtrack
if ( $F = \emptyset$ ) then  $P$  is solution
else
   $x_i := \text{select\_var}(F)$ 
   $\Pi := \text{domain\_partition}(d_i)$ 
  for all  $\pi_k \in \Pi$  do
     $D' := \text{look\_ahead}_w(x_i, \pi_k, F, D)$ 
    if ( $|\pi_k| = 1$ ) then /* strong assign. */
       $FC_w(P \cup \{x_i := \pi_k\}, F - \{x_i\}, D')$ 
    else /* weak assignment */
       $d'_i = \pi_k$ 
       $FC_w(P, F, D')$ 
EndProcedure

Function  $\text{look\_ahead}_w(x_i, \pi_k, F, D)$ 
for all  $x_j \in F$  do if  $x_j \neq x_i$  then
  for all  $a \in d_j$  do
     $d'_j = \emptyset$ 
    for all  $b \in \pi_k$  do
      if (consistent( $x_i, x_j, b, a$ )) then  $d'_j = d'_j \cup \{a\}$ 
Return ( $D'$ )

```

Figure 3. Forward checking with weak assignments.

### 5 Experimental Results

To evaluate the practical validity of our approach a set of experiments was devised using *crossword puzzles*. A crossword puzzle is characterized by a set of words (called dictionary) and a grid that may have some of its cells labelled as void. Consecutive non void cells (horizontally or vertically) define a slot. The task is to place a word to each horizontal slots in such a way that vertical slots form dictionary words, too ([Ginsberg *et al.*, 1990]). They are of practical interest because they can be seen as an abstraction of configuration tasks where the components of an assembly have to be decided in such a way that interdependency restrictions are fulfilled.

Crossword puzzles can be represented as binary CSPs with variables associated to grid slots. For each slot, its domain is the set of dictionary words having the appropriate length. Intersecting slots are constrained, restricting pairs of words to those matching in their intersection.

In our experiments we used random puzzles with square grids and without void cells. In addition, we tightened constraints preventing rows being equal from columns in order to avoid symmetric solutions (they were found to make problem instances significantly easier).

Thus, our random puzzles are characterized by a tuple  $\langle l, m, v \rangle$  where  $l$  is the words length and the grid dimension,  $m$  is the dictionary cardinality and  $v$  is the alphabet cardinality. Random instances were generated by selecting  $m$  words out of the  $v^l$  choices using a uniform probability distribution. We experimented on the following problem classes:  $\langle 5, 20-99, 5 \rangle$ ,  $\langle 7, 20-62, 3 \rangle$  and  $\langle 9, 15-31, 2 \rangle$  ( $m$  was used as a varying parameter). For each parameter setting, samples of 50 random instances were generated forming a total of 7,000 instances. We observed that, for small  $m$ , problems are overconstrained and do not have any solution. If  $m$  is increased, there is a point after which problems become abruptly underconstrained and the number of

solutions grows very fast. If only one solution is sought, this point corresponds to a peak on average problem difficulty similar to that observed in other domains [Cheeseman *et al.*, 1991]. In our three problem classes the rank for  $m$  was chosen to coincide with the peak. Therefore, our experiments were on *hard* instances having few, or no solution.

An interesting feature of crossword puzzles is that constraints are implicit in the letters forming the words. Words having many matching letters behave similarly in the problem. More precisely, the pruning effect of assigning two words to a slot is exactly the same with respect to slots intersecting in their matching letters and totally different with respect to slots intersecting in their different letters. In our Implementation of  $FC_w$  for crossword puzzles, the *domain\_partition* function (Fig. 3) partitions domains into groups where every word differs from the rest in at most  $\delta$  letters. Delta was set to one for the first class of problems ( $l=5$ ) and to two for the other two classes ( $l=7$  and  $l=9$ ). These values for  $\delta$  were obtained by sampling the dictionaries and observing that more similarity rarely occurred and less similarity gave a too coarse partition.

Each problem instance was solved using FC and  $FC_w$  finding one and all solutions. In all cases the *minimum remaining values* variable ordering heuristic was used [Haralick and Elliott, 1980]. Figure 4 summarizes the results of our experiment. We report average computational effort versus dictionary size for finding one and all problem solutions. Computational effort is given as the number of consistency checks (a consistency check occurs each time that the algorithm evaluates the consistency of assigning two words to two constrained variables). The number of consistency checks is a more appropriate measure for computational effort than the number of visited nodes, since visiting some nodes may require more computation than others. However, we observed a strong correlation between these two parameters.

Each graph in Figure 4 corresponds to a problem class and provides the information for the two tasks under consideration. It can be observed that, for the three problem classes, through the whole rank of  $m$ , and for the two tasks,  $FC_w$  clearly outperforms FC. The gain of  $FC_w$  goes up to a factor of 2.3 (up to 3.1 and 2.0 in terms of visited nodes and CPU time, respectively). The greatest gains are obtained on the hardest instances for each task. That is, at the peak if one solution is wanted, and for the largest dictionaries if all solutions are wanted.

It should be noted that  $FC_w$  increases its advantage as the words length increases and the alphabet cardinality decreases. An explanation is that, for longer words, search trees are deeper and similar problems have longer common paths. Thus, it is more profitable merging similar problems. In addition, with smaller alphabets similar words are more likely to occur. Therefore, weak assignments of larger groups of words can be performed.

## 6 Related Work

The importance of values having similar behavior in a problem was stressed in [Freuder, 1991] and related to the

concept of value *interchangeability*. Some forms of local interchangeability have been found especially useful. Two values are *neighborhood interchangeable* if they are inconsistent with *exactly* the same values of the rest of variables. Assigning neighborhood interchangeable values to a variable results on identical subproblems. In [Haselbock, 1993] an algorithm which merges search branches of neighborhood interchangeable values is presented. In addition, neighborhood interchangeability with respect single constraints is used to save consistency checks. Neighborhood interchangeability is a too strong condition for some real problems. For this reason weaker forms of interchangeability were defined. One value  $a$  is *neighborhood substitutable* for another value  $b$  if the set of values inconsistent with  $a$  is a subset of the values inconsistent with  $b$ . Assigning neighborhood substitutable values produces subproblems one included in the other. In [Bellicha *et al.*, 1994], algorithms exploiting neighborhood substitutability are developed. Neighborhood substitutability is still a strong condition. For instance, in the crossword domain there is not neighborhood substitutability (two different words have a completely different behaviour in, at least, one constraint). In this domain, neither Haselbock's algorithm, nor Bellicha's algorithm produce any tree reduction.

Our work extends this line of research in the sense that it does not require values to be *exactly* neighborhood interchangeable or substitutable to avoid unnecessary search.  $FC_w$  would certainly be cost effective in those cases, but it can also obtain tree reductions when values have a *not so structured* similar behaviour, or when values are close to accomplish interchangeability or substitutability.

Avoiding unnecessary duplication of search has been the subject of intense research. In [Frost and Detchter, 1994], unsolvable subproblems found during search are recorded as nogoods and they are skipped if they are found again in the future. Alternatively, [Freuder and Hubbe, 1995] present a method for decomposing subproblems and propose an algorithm exploiting it. Roughly, when a subproblem is to be solved, its intersection with one of its previous siblings is computed and factored out. As in our approach, [Freuder and Hubbe, 1995] only consider similarities among sibling nodes, since they only differ in the effect of one assignment and are more likely to be similar. However, our approach is different in spirit. While their algorithm tries to detect similarities with previously unsuccessfully solved subproblems,  $FC_w$  groups similar subproblems and merges them into a single problem. A performance comparison between these two methods in different domains remains as future work.

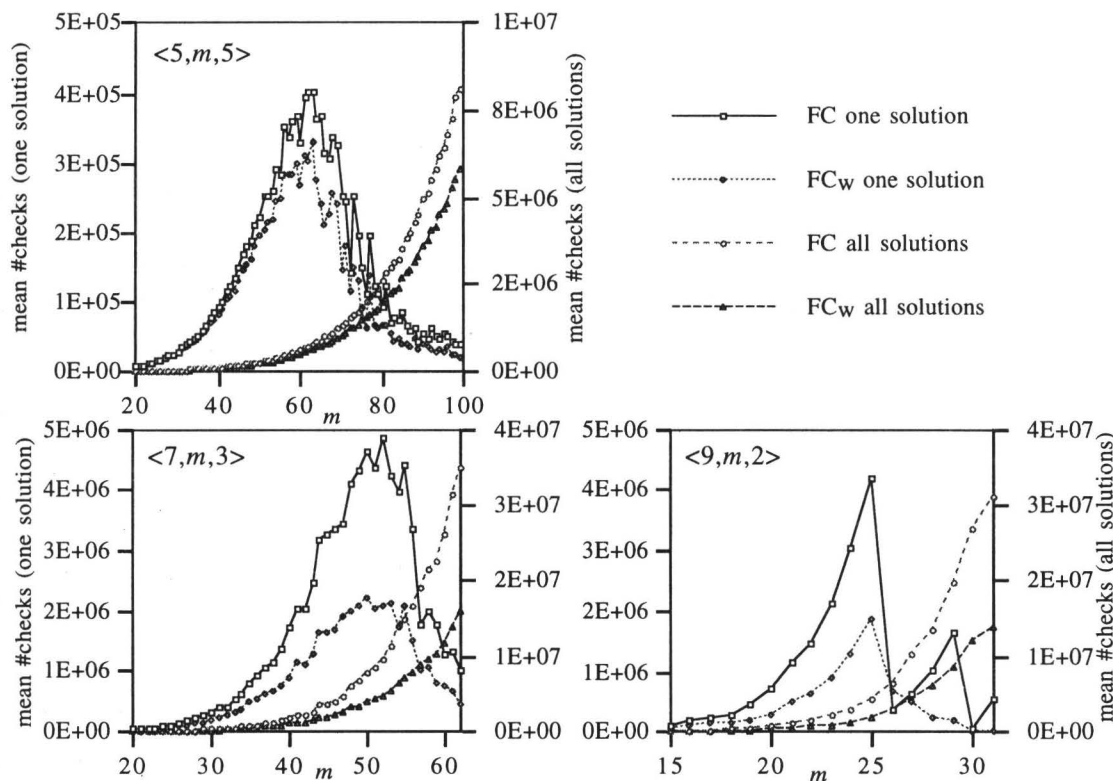


Figure 4. Experimental results on three classes of *random crossword puzzles*. Each graph reports mean computational effort (in number of consistency checks).

## 7 Conclusion

We have introduced a method, based in the concept of weak assignment, for merging constraint satisfaction subproblems. It avoids some search redundancy associated to subproblem similarities. We have incorporated this method to forward checking for the tasks of finding one and all problem solutions. Our experiments on hard random crossword problems show that forward checking can increase significantly its performance when enhanced with merging capabilities.

Our method extends previous work on value interchangeability in the sense that it requires weaker conditions for subproblem similarity. It can be combined with other CSP algorithms different from FC, keeping a higher consistency level among future variables. The study of these combinations and their applicability to other domains are open questions requiring further research.

## Acknowledgments

I am in debt to Pedro Meseguer for his help on carrying out this research. I thank Ricard Gavaldà for valuable comments on an earlier version of this paper.

## References

[Bellicha *et al.*, 1994] A. Bellicha, C. Capelle, M. Habib, T.

- Kökény, M. Vilarem. CSP techniques using partial orders on domain values. In *ECAI-94 Workshop on CSP issues raised by practical applications*, 47-56, 1994.
- [Cheeseman *et al.*, 1991] P. Cheeseman, B. Kanefsky, W.M. Taylor. Where the really hard problems are. In *Proceedings of IJCAI-91* pages 331-337, 1991.
- [Freuder, 1991] E. C. Freuder. Eliminating interchangeable values in constraint satisfaction problems. In *Proceedings of AAAI-91* pages 227-233, 1991.
- [Freuder and Hubbe, 1995] Eugene C. Freuder and Paul D. Hubbe. Extracting constraint satisfaction subproblems. In *Proceedings of IJCAI-93*, pages 548-555, 1993.
- [Frost and Detchter, 1994] Daniel Frost and Rina Detchter. Dead-end driven learning. In *Proceedings of AAAI-94*, pages 294-300, 1994.
- [Ginsberg *et al.*, 1990] Matthew L. Ginsberg, Michael Frank, Michael P. Halpin, and Mark C. Torrance. Search lessons learned from crossword puzzles. In *Proceedings of AAAI-90*, pages 210-215, 1990.
- [Haralick and Elliott, 1980] R. M. Haralick and G. J. Elliot. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263-313, 1980.
- [Haselbock, 1993] Alois Haselbock. Exploiting Interchangeabilities in constraint satisfaction problems. In *Proceedings of IJCAI-93*, pages 282-287, 1993.