Retroactive Ordering for Dynamic Backtracking

Roie Zivan ZIVANR@CS.BGU.AC.IL
Uri Shapen SHAPENKO@CS.BGU.AC.IL
Moshe Zazone MOSHEZAZ@CS.BGU.AC.IL
Amnon Meisels AM@CS.BGU.AC.IL
Department of Computer Science,
Ben-Gurion University of the Negev,
Beer-Sheva, 84-105, Israel

Abstract

Dynamic Backtracking (DBT) is a well known algorithm for solving Constraint Satisfaction Problems. In DBT, variables are allowed to keep their assignment during backjump, if they are compatible with the set of eliminating explanations. A previous study has shown that when DBT is combined with variable ordering heuristics it performs poorly compared to standard Conflict-directed Backjumping (CBJ) (Baker, 1994). The special feature of DBT, keeping valid elimination explanations during backtracking, can be used for generating a new class of ordering heuristics. In the proposed algorithm, the order of already assigned variables can be changed. Consequently, the new class of algorithms is termed *Retroactive DBT*.

The proposed algorithm exploits the fact that when the assignment of a variable is complete, its heuristic evaluation can be higher than variables which were assigned before it. For the min-domain heuristic, the newly assigned variable can be moved to a position in front of assigned variables with larger domains and as a result prune the search space more effectively. The experimental results presented in this paper show an advantage of the new class of heuristics and algorithms over standard DBT and over CBJ on two different problem scenarios: random problems and realistic structured problems. All algorithms tested were combined with forward-checking and used a *Min-Domain* heuristic.

1. Introduction

Conflict Based Backjumping (CBJ) is a technique which is known to improve the search of Constraint Satisfaction Problems (CSPs) by a large factor (Dechter, 2003; Kondrak & van Beek, 1997; Chen & van Beek, 2001). Its efficiency increases when it is combined with forward checking (Prosser, 1993). The advantage of CBJ over standard backtracking algorithms lies in the use of conflict sets in order to prune unsolvable sub search spaces. Conflicts which caused a removal of values from a variables' domain are stored in the variables' conflict set. When a dead end is detected only variables whose assignment is included in the conflict set of the backtracking variable need to be considered as a target for backtrack (we say that the algorithm backtracks to the *culprit* variable/assignment (Prosser, 1993)). The down side of CBJ is that when such a backtrack (backjump) is performed, assignments of variables which were assigned later than the culprit assignment are discarded.

Dynamic Backtracking (Ginsberg, 1993) improves on standard CBJ by preserving assignments of non conflicting variables during back-jumps. In the original form of DBT, the culprit variable which replaces its assignment is moved to be the last among the assigned variables. In other words,

the new assignment of the culprit variable must be consistent with all former assignments (Ginsberg, 1993).

Although DBT saves unnecessary assignment attempts and was proposed as an improvement to CBJ, a later study by Baker (Baker, 1994) has revealed a major drawback of DBT. According to Baker, when no specific ordering heuristic is used, DBT performs better than CBJ. However, when ordering heuristics which are known to improve the run-time of CSP search algorithms are used (Haralick & Elliott, 1980; Bessiere & Regin, 1996; Dechter & Frost, 2002), the performance of DBT is slower than the performance of CBJ. This phenomenon is easy to explain. Whenever the algorithm performs a back-jump it actually takes a variable which was placed according to the heuristic in a high position and moves it to a lower position. Thus, while in CBJ, the variables are ordered according to the specific heuristic, in DBT the order of variables becomes dependent on the algorithm's behavior (Baker, 1994).

In order to leave the assignments of non conflicting variables without a change on backjumps, DBT maintains a system of eliminating explanations ($Nogoods$) (Ginsberg, 1993). As a result, the DBT algorithm maintains dynamic domains for all variables and can potentially benefit from the *Min-Domain* (fail first) heuristic.

The present paper investigates a number of improvements to DBT that use radical versions of the *Min-Domain* heuristic.

1. The algorithm avoids moving the culprit variable to the lowest position in the partial assignment. This alone can be enough to eliminate the phenomenon reported by Baker (Baker, 1994).

2. The assigned variables which were originally ordered in a lower position than the culprit variable can be reordered according to their current heuristic value (for example, the size of there current domain).

3. A $retroactive$ ordering heuristic in which assigned variables are reordered is proposed. A *retroactive* heuristic allows an assigned variable to be moved upwards beyond assigned variables as far as the heuristic is justified.

Consider the run of a search algorithm in which variables are ordered according to the *Min-Domain* heuristic. Although variables are selected according to a *Min-Domain* heuristic, a newly assigned variable can have a smaller current domain than previously assigned variables. This can happen because of two reasons.

1. As a result of *forward-checking* which might cause values from the current variables' domain to be eliminated due to conflicts with unassigned variables.

2. As a result of multiple backtracks to the same variable which eliminate at least one value each time.

The present study proposes to exploit the heuristic properties, not only by selecting the next variable to be assigned, but by placing it in its *right* place among the already assigned variables after it is assigned successfully.

The combination of the three ideas above was found to be successful in the empirical study presented in the present paper.

Constraints satisfaction problems (CSPs) are presented in Section 2. Next, a description of the dynamic backtracking (DBT) algorithm is presented, followed by an explanation of how *Forward Checking* can be introduced into DBT. The proposed *Retroactive FC_DBT* algorithm is described in Section 4. A correctness proof for *Retroactive FC_DBT* is presented in Section 5. The experimental section includes an empirical study which compares CBJ, DBT and *Retroactive* DBT with and without forward-checking. The algorithms were evaluated on two different scenarios: random CSPs and a realistic structured problems (meeting scheduling (Wallace & Freuder, 2002; Modi & Veloso, 2004; Maheswaran, Pearce, Bowring, Varakantham, & Tambe, 2006)). Retroactive DBT was found to perform better on all these three experimental scenarios. On realistic meeting scheduling problems the improvement of *Retroactive DBT* is by a large factor.

2. Constraint Satisfaction Problems

A *Constraint Satisfaction Problem* (*CSP*) is composed of a set of n variables $V_1, V_2, ..., V_n$. Each variable can be assigned a single value from a discrete finite domain. Constraints or **relations** R are subsets of the Cartesian product of the domains of constrained variables. For a set of constrained variables $\{V_i, V_j, ..., V_m\}$, with domains of values for each variable $\{D_i, D_j, ..., D_m\}$, the constraint is defined as $R \subseteq D_i \times D_j \times ... \times D_m$. A binary constraint R_{ij} between any two variables V_j and V_i is a subset of the Cartesian product of their domains; $R_{ij} \subseteq D_j \times D_i$.

An assignment (or a label) is a pair $\langle var, val \rangle$, where var is a variable and val is a value from var's domain that is assigned to it. A *partial solution* is a consistent set of assignments of values to a set of variables. A **solution** to a *CSP* is a partial solution that includes assignments to all variables (Dechter & Frost, 2002)..

3. Dynamic Backtracking

The dynamic backtracking (DBT) algorithm is presented following (Baker, 1994). We assume in our presentation that the reader is familiar with CBJ (Prosser, 1993).

Like any backtrack algorithm, DBT attempts to extend a *partial solution*. A partial solution is an ordered set of value assignments to a subset of the CSP variables which is consistent (i.e. violates no constraints). The algorithm starts by initializing an empty partial solution and then attempts to extend this partial solution by adding assigned variables to it. When the partial solution includes assignments to all the variables of the CSP, the search is terminated successfully.

In every step of the algorithm the next variable to be assigned is selected according to the heuristic in use, and the values in its current domain are tested. If a value is in conflict with a previous assignment in the partial solution, it is removed from the current domain and is stored together with its eliminating $Nogood$. Otherwise, it is assigned to the variable and the new assignment is added to the partial solution (Baker, 1994; Ginsberg, 1993).

An order is defined among the assignments in the partial solution. In the simplest form, this order is simply the order in which the assignments were performed (other options will be discussed).

Following (Ginsberg, 1993; Baker, 1994), all $Nogoods$ are of the following form:

$(v_1 = q_1) \wedge ... \wedge (v_{k-1} = q_{k-1}) \Rightarrow v_k \neq q_k$

The left hand side serves as the explanation for the invalidity of the assignment on the right hand side. An eliminating $Nogood$ is stored as long as its left hand side is consistent with the current

partial solution. When a $Nogood$ becomes invalid, it is discarded and the forbidden value on its right hand side is returned to the current domain of its variable.

When a variable's current domain empties, the eliminating $Nogoods$ of all its removed values are resolved and a new $Nogood$ which contains the union of all assignments from all $Nogoods$ is generated. The new $Nogood$ is generated as follows. The right hand side of the generated $Nogood$, includes the assignment which was ordered last in the union of all $Nogoods$ (the culprit assignment). The left hand side is a conjunction of the rest of the assignments in the united set (Ginsberg, 1993; Baker, 1994).

After the new $Nogood$ is generated, all $Nogoods$ of the backtracking variable which include the culprit assignment are removed and the corresponding values are returned to the current domain of the backtracking variable. Notice that this assignment to the backtracking variable cannot possibly be in conflict with any of the assignments which are ordered before the culprit assignment. Otherwise, they would have been included in an eliminating $Nogood$. The culprit variable on the right hand side of the generated $Nogood$ is the next to be considered for an assignment attempt, right after its newly created $Nogood$ is stored. Its new position in the order of the partial solution is after the latest assignment (i.e. it is moved to a lower place in the order than the one it had before). Note again that variables that were originally assigned after the culprit variable can stay assigned.

3.1 DBT with Forward Checking

Forward Checking is a common method for maintaining consistency in CSP search (Prosser, 1993; Kondrak & van Beek, 1997; Dechter, 2003). After each assignment, all values in the domains of unassigned variables which are in conflict with the new assignment are removed. Thus the domains of unassigned variables include only values which are consistent with the current partial solution. An empty domain of an unassigned variable triggers a backtrack operation. When an assignment is replaced as a result of a backtrack, all values which were removed from the domains of unassigned variables due to a conflict with the replaced assignment must be returned to the current domain of their variables (Prosser, 1993).

Forward Checking introduces two advantages to the search. The first is an early detection of an inconsistent assignment. The second is the fact that the relevant size of domains of variables can be easily computed. This makes the *Min Domain* heuristic very efficient (Dechter & Frost, 2002).

CBJ with *Forward Checking* (FC_CBJ) is known to be a successful algorithm (Prosser, 1993; Kondrak & van Beek, 1997). In order to introduce *Forward Checking* into *Dynamic Backtracking*, after each assignment all values of unassigned variables are checked and inconsistent values are removed and stored along with their eliminating explanations. When an empty domain is encountered, the algorithm operates similarly to the case for any exhausted domain. A new $Nogood$ is resolved out of the stored $Nogoods$ of the variable whose domain was emptied. The culprit assignment is removed from the partial solution after the new $Nogood$ is stored for that variable. After an assignment is replaced, all eliminating $Nogoods$ which include the replaced assignment are discarded and the corresponding values are returned to their variable's domain.

Figure 1 presents the code of *Forward Checking Dynamic Backtracking* (FC_DBT). The main procedure of the algorithm attempts to extend the current partial solution which is the assignment $1..pos-1$ to $1..pos$. It ends when all the variables are assigned.

```
FC_DBT
1.  var_list ← variables;
2.  assigned_list ← φ;
3.  pos ← 1;
4.  while (pos < N)
5.     next_var ← select_next_var(var_list);
6.     var_list.remove(next_var);
7.     assigned_list.insert(next_var, pos);
8.     assign(next_var);
9.  report solution;

assign(var)
10. for each (value ∈ var.current_domain)
11.    var.assignment ← value;
12.    consistent ← true;
13.    forall (i ∈ var_list
          and while consistent)
14.       consistent ← check_forward(var, i);
15.    if not (consistent)
16.       nogood ← resolve_nogoods(i);
17.       store(var, nogood);
18.       undo_reductions(var, pos);
19.    else
20.       pos ← pos+1;
21.       return;
22. var.assignment ← Nil;
23. backtrack(var);

backtrack(var)
24. nogood ← resolve_nogoods(var);
25. if (nogood = φ)
26.    report no solution;
27.    stop;
28. culprit ← nogood.RHS_variable;
29. store(culprit, nogood);
30. undo_reductions(culprit, pos_culprit);
31. culprit.assignment ← Nil;
32. var_list.insert(var);
33. assigned_list.remove(var, pos);
34. pos ← pos-1;
35. forall (j, j ∈ assigned_list and
       pos_j > pos_culprit and pos_j ≤ pos-1)
36.       consistent ← check_forward(j, culprit);
37.       if not (consistent)
38.          backtrack(culprit);
39.          return;
40. var_list.insert(culprit);
41. assigned_list.remove(culprit, pos_culprit);
```

Figure 1: The FC_DBT algorithm with the original ordering of DBT (Ginsberg, 1993)

Picking the next variable for assignment among the unassigned variables is performed by the procedure $select\_next\_var$ according to the ordering heuristic. Assignments are made by calling the procedure **assign** (line 8).

Procedure **assign** is passed the variable for assignment. Each of the values in the current domain is considered for assignment. If the value is consistent with the current domain of all unassigned variables (lines 12-14) the assignment is completed successfully, and the value of pos is incremented to extend the partial solution.

If the value is inconsistent, it is stored along with its eliminating $Nogood$. The $Nogood$ is obtained from the variable whose domain was emptied (lines 16, 17). Values that were removed from unassigned variables because of a direct conflict with this value, are returned back by procedure **undo_reductions**(var, pos_var) (line 18).

If a domain is exhausted, procedure **backtrack** is called, procedure **backtrack** is passed the variable whose domain was emptied. The eliminating $Nogoods$ of the removed values from the domain of that variable are resolved into a new $Nogood$ (line 24). If the generated $Nogood$ is empty, the algorithm reports *no solution* and terminates (lines 25-27). Otherwise, the assignment of

check_forward(var, i)
1. **for** each val $\in$ i.current_domain
2. **if not** check(i, val, var.assignment)
3. remove val from i.current_domain
4. nogood $\leftarrow$ (var = var.assignment $\rightarrow$ i $\neq$ val)
5. store(i, nogood)
6. **return** (i.current_domain $\neq \phi$)

undo_reductions(var, pos_var)
7. **forall** var_1 $\in$ assigned_list **and** pos_var_1 $\leq$ pos)
8. **remove** eliminating nogoods of var_1 containing var
9. **forall** (var_2 $\in$ assigned_list **and** pos_var_2 $>$ pos_var)
10. **check_forward**(var_2, var_1)
11. **forall** var_1 $\in$ var_list
12. **remove** eliminating nogoods of var_1 containing var
13. **forall** (var_2 $\in$ assigned_list **and** pos_var_2 $>$ pos_var)
14. **check_forward**(var_2, var_1)

Figure 2: **check_forward**() and **undo_reductions**() for FC_DBT

the variable that is on the right hand side of the generated $Nogood$ is removed and stored with the generated $Nogood$ as its eliminating explanation (lines 28, 29). Next, all $Nogoods$ which contained the removed assignment are discarded (line 30), the current partial solution size is reduced by one and the backtracking variable is entered into the list of unassigned variables (lines 31-34). Now comes a special part of the algorithm, that deals with the inherent dynamic ordering of FC_DBT. The current_domain of the culprit variable has to be updated by forward checking it against the variables that were assigned **after** it. This is done in lines 35-36. In case the current_domain of the culprit variable empties, procedure **backtrack** is called (lines 37-39). In case the culprit agent remains consistent at the end of the process, it is removed from the assigned list and entered into the unassigned set (lines 40,41).

The procedure **check_forward** (presented in Figure 2) is similar to the one introduced in (Prosser, 1993). It is responsible for the removal of conflicting values from the domain of all unassigned variables. It returns false if a variable's domain is emptied.

The procedure **undo_reductions** (also presented in Figure 2) is called as a result of removing an assignment of the passed variable var. Values whose eliminating explanations include the removed assignment are returned to their variable's current_domain. Returned values must be filtered in case they are in conflict with assignments with lower position than var (i.e ordered *after* var).

4. Retroactive Dynamic Backtracking

The first step in enhancing the heuristic (*Min-Domain* in our case) for DBT is to avoid the move forward in the resulting order, of variables that the algorithm backtracks to (i.e. culprit variables). One way to do this is to try to replace the assignment of the culprit variable and *leave the variable*

in the same position. This of course would require to check that all assignments of later variables, in the partial solution, are consistent with the replaced assignment. Since the replaced assignment is in a higher position, in case of a conflict, the assignment of the lower position variable is the one to be replaced.

The second potential enhancement is to reorder the assigned variables that have a lower order than the culprit assignment which was replaced. This step takes into consideration the possibility that the replaced assignment of a variable that lies higher in the order has the potential to change the size of the current domains of the already assigned variables that are ordered after it. The simplest way to perform this step is to reorder these variables by the selected heuristic.

The third enhancement derives from the observation that in many cases the size of the current domain of a newly assigned variable is smaller than the current domains of variables which were assigned before it. There can be two reasons for this phenomenon.

1. During the assignment attempt, some values may be in conflict with values of unassigned variables. These values may be pruned as a result of *Forward Checking* (e.g. causing an empty domain of an unassigned variable). As a result, by the time a consistent assignment is found for the selected variable, the size of its current domain can be smaller than at the beginning of the assignment procedure.

2. Smaller domains during the assignment procedure may be the result of backtracking. On backtracks, the current domain of a variable whose domain was exhausted is in many cases small. Only values that were eliminated by the culprit assignment are returned to its domain.

Allowing a reordering of assigned variables enables the use of heuristic information which was not available while the previous assignments have been performed. This takes heuristics which take the current domain size of variables into consideration (Haralick & Elliott, 1980; Bessiere & Regin, 1996; Dechter & Frost, 2002) to a new level and generates a radical new approach. Variables can be moved up in the order, in front of assigned variables of the partial solution. As long as the new assignment is placed *after* the most recent assignment which is in conflict with one of the variable's values, the size of the domain of the assigned variable is not changed. Such a heuristic is *retroactive* since variables are moved in front of assigned variables which in any standard heuristic would not have been touched. The algorithm can check the sizes of the domains of variables with lower priority than the last assignment in the LHS of the generated $Nogood$ and move the backtracking variable to a higher position than variables with larger domains.

In the best ordering heuristic proposed by the present paper, the new position of the assigned variable in the order of the $partial\_solution$ is dependent on the size of its current domain. The heuristic checks all assignments from the last up to the first assignment which is included in the union of the newly assigned variable's eliminating Nogoods. The new assignment will be placed right after the first assigned variable with a smaller current domain.

Figure 5 presents an example of this ordering heuristic for *Retroactive DBT*. In the first step the variables are ordered lexicographically. Beneath each variable is the size of its current domain. Notice that although variables are ordered according to the *Min-Domain* heuristic, the size of domains is dynamic throughout the search. Therefore, in this example the size of the current domain of variable V_4 is smaller than the size of variables V_1 and V_2. In the second step, variable V_6 is assigned and its current domain size becomes 2. Assume that the union of the eliminating $Nogoods$ of V_6 includes the assignment of V_1 and V_3. Now, the algorithm searches for a new position for V_6.

Retroactive FC_DBT

```
1.  var_list ← variables;
2.  assigned_list ← φ;
3.  pos ← 1;
4.  while (pos < N)
5.     next_var ← select_next_var(var_list);
6.     var_list.remove(next_var);
7.     assign(next_var);
8.  report solution;

procedure assign(var)
10. for each (value ∈ var.current_domain)
11.    var.assignment ← value;
12.    consistent ← true;
13.    forall (i ∈ var_list)
          and while consistent
14.       consistent ← check_forward(var, i);
15.    if not (consistent)
16.       nogood ← resolve_nogoods(i);
17.       store(var, nogood);
18.       undo_reductions(var, pos);
19.    else
20.       nogood ← resolve_nogoods(pos);
21.       lastVar ← nogood.RHS_variable;
21.       newPos ← select_new_pos(var, lastVar);
22.       assigned_list.insert(var, newPos);
23.       forall (var_1 ∈ assigned_list) and (pos_var_1 > newPos)
24.          check_forward(var, var_1);
25.          update_nogoods(var, var_1);
26.       forall (var_2 ∈ var_list)
27.          update_nogoods(var, var_2);
28.       pos ← pos+1;
29.       return;
30. var.assignment ← Nil;
31. backtrack(var);
```

Figure 3: The Retroactive FC_DBT algorithm

```
procedure backtrack(var)
1.  nogood ← resolve_nogoods(var);
2.  if (nogood = φ)
3.     report no solution;
4.     stop;
5.  culprit ← nogood.RHS_variable;
6.  store(culprit, nogood);
7.  culprit.assignment ← Nil;
8.  undo_reductions(culprit, pos_culprit);
9.  forall (var_1 ∈ assigned_list) and (pos_var_1 > newPos)
10.    undo_reductions(var_1, pos_var_1);
11.    var_1.assignment ← Nil;
12.    var_list.insert(var_1);
13.    assigned_list.remove(var_1);
14. pos ← pos_culprit;

procedure update_nogoods(var_1, var_2)
15. for each (val ∈ {var_2.domain - var_2.current_domain})
16.    if not (check(var_2, val, var_1.assignment))
17.       nogood ← remove_eliminating_nogood(var_1, val);
18.       if not (∃var_3 ∈ nogood and pos_var_3 < pos_var_1)
19.          nogood ← ⟨var_1.assignment → var_2 ≠ val⟩;
20.       store(var_2, nogood);
```

Figure 4: Retroactive FC_DBT algorithm (continued)

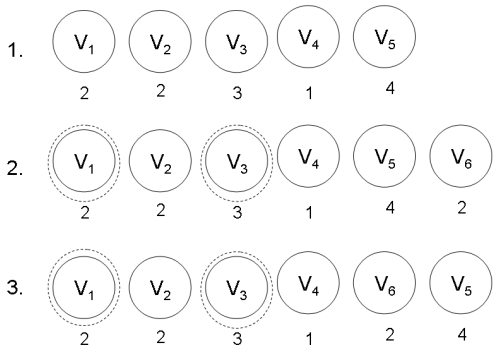

Figure 5: Example of Reordering after an assignment.

It starts by comparing its current domain size to the current domain size of V_5. Since the current domain of V_6 is smaller it is compared with the current domain size of V_4 and is found to be larger.

Thus, the state in step 3 is achieved where V_6 is placed after V_4 and before V_5. Now, due to the assignment of V_6, forward checking has to be performed for V_5 which is an assigned variable.

Figures 3, 4 present the code of *Retroactive Forward Checking Dynamic Backtracking* ($Retro\_FC\_DBT$). The main procedure of the algorithm is identical to standard FC_DBT.

Procedure **assign** is changed to support retroactive heuristics. After a successful consistency check (lines 12-14), the variable is moved to a new position. The new position is returned by calling **select_new_position**($var, lastVar$). This function selects the highest position for the assigned variable according to the specified heuristic and as long as it is lower than the position of the most recent variable appearing in its stored Nogoods ($lastVar$) (lines 20-21). The change in the position of the variable that is being assigned generates a need to update the domains of future variables (lines 23-27) However, this update can not lead to inconsistency since the current assignment of these variables are not in conflict with the new assignment. These future variables include variables that kept their assignment.

The procedure **update_nogoods** is called to remove values in conflict with the newly placed variable. These values may have already been pruned by $Nogoods$ that now contain lower order variables and must be updated to be eliminated by the variable that is currently being assigned.

In Procedure **backtrack** (Figure 4), as in standard FC_DBT, the $Nogood$ is resolved and checked for termination (lines 1-4). Then the algorithm unassigns the culprit variable and variables later in the order than the culprit, and returns removed values to their domains (lines 9-13). At the end of the backtrack procedure pos is adjusted to a value that extends the actual assignment (line 14).

The change from the standard algorithm is in lines (9-13). Assignments that are placed after the culprit in the order are removed and their assigned values are restored to the variables' domains. After each removal of an assignment, the $Nogoods$ of lower order variables are updated. $Nogoods$ containing the removed assignment are discarded and their corresponding values are returned to the variables' domains.

Introducing *Forward Checking* into *Retroactive DBT* is more complicated than in the case of standard DBT. After an assignment is performed all inconsistent values must be removed not only from the domains of unassigned variables but also from the domains of assigned variables with a lower priority than the new assignment.

In the best ordering heuristic proposed by the present paper, the new position of the assigned variable in the order of the $partial\_solution$ is dependent on the size of its current domain. The heuristic checks all assignments from the last up to the first assignment which is included in the union of the newly assigned variable's eliminating Nogoods. The new assignment will be placed right after the first assigned variable with a smaller current domain.

5. Correctness of Retroactive DBT

We first assume the correctness of the standard DBT algorithm (as proven in (Ginsberg, 1993)) and prove that after the changes made for forward checking and for retroactive heuristics, it is still sound, complete and it terminates.

Soundness is immediate since after each successful assignment the $partial\_solution$ is consistent. Therefore, when the $partial\_solution$ includes an assignment for each variable the search is terminated and a consistent solution is reported. Note that lines 23-25 take care of the retroactive part of forward checking. □

As in the case of standard *DBT*, the completeness of *Retroactive DBT* derives from exploring the entire search space except for sub search spaces which were found not to contain a solution. One needs to prove that the sub search spaces which *DBT* does not search do not contain solutions. Sub search spaces are pruned by $Nogoods$. It is enough to prove the consistency of the set of $Nogoods$ generated by *Retroactive DBT*.

In other words, that the assignment of values removed by $Nogoods$ never lead to solutions. For standard *DBT* this is proven in the original paper (Ginsberg, 1993). The consistency property of $Nogoods$ generated by *FC_DBT* can be shown as follows. First, observe that during forward-checking (function **check_forward** in Figure 2) $Nogoods$ are standardly stored as explanations to removed values in the future variable's domain. Next, consider the case of a backtrack triggered by lines 15-18 and 22-23 of the main *FC_DBT* procedure in Figure 1. It is easy to see that $Nogoods$ of the future variables are resolved identically to those of standard DBT. this proves the completeness of *FC_DBT*.

The consistency of $Nogoods$ generated by *Retroactive FC_DBT* follows immediately from the same property in *FC_DBT*. The only difference is that future variables against which consistency are checked by **check_forward**(), can be already assigned. This special case is treated in lines 23-25 of the main function of *Retroactive FC_DBT* in Figure 3. Clearly, all $Nogoods$ resolution are identical to standard *FC_DBT*. This proves the completeness of *Retroactive FC_DBT*.

Last, we need to prove that the algorithm terminates. In order to do so we prove the following lemmas:

Lemma 1 *The number of times that a pair of variables can change order between them (i.e. one moving in front of the other) without another variable in a higher position of both of them replacing its assignment is bounded by* $2m$ *where m is the initial domain size.*

proof: variables change places when their assignment is replaced. An assignment replacement includes the removal of the assign value in accordance to its eliminating $Nogood$. Since variables are not moved in front of assignments which are included in their nogoods variable X_i can move in front of variable X_j all the nogoods of X_i must be with a higher position than X_j. Therefore, as long as non of the variables in a higher position change their assignment, non of the values removed from the variables domains can be restored in the variables domains, thus the number of times the variables can change places is bound by the size of their domains. □

Lemma 2 derives directly from Lemma 1.

Lemma 2 *The number of position replacements of variables which are in a position lower than position* k *is bounded.*

Therefore, for $k = 0$, the number of position replacements in the entire CSP is bounded. Thus, the termination of the algorithm derives from the termination of static backtrack algorithms. □

6. Experimental Evaluation

The common approach in evaluating the performance of *CSP* algorithms is to measure time in logical steps to eliminate implementation and technical parameters from affecting the results. Two measures of performance are used by the present evaluation. The number of assignments and the number of constraints checks (Prosser, 1996; Kondrak & van Beek, 1997).

The experiments were conducted on two problem scenarios: Random *CSPs* and on structured problems that represent a realistic scenario - Meeting Scheduling Problems (Gent & Walsh, 1999).

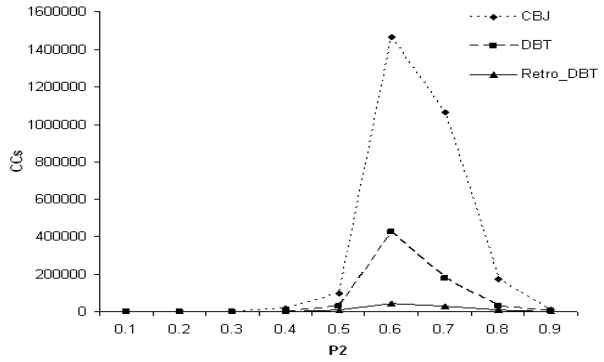

Figure 6: Constraints checks performed by *DBT*, *CBJ* and *Retroactive DBT* ($p_1 = 0.3$).

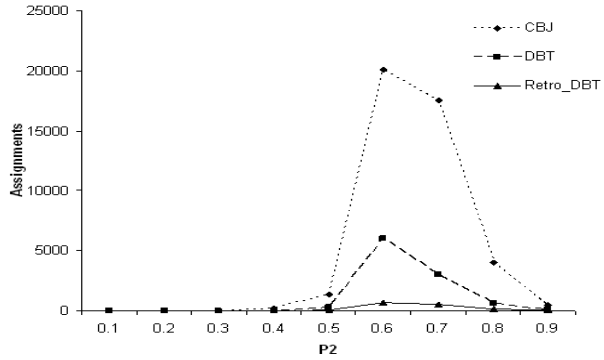

Figure 7: Assignments performed by *DBT*, *CBJ* and *Retroactive DBT* ($p_1 = 0.3$).

6.1 Experiments on Random CSPs

Random *CSPs* are parametrized by n variables, k values in each domain, a constraints density of p_1 and a tightness p_2 which are commonly used in experimental evaluations of CSP algorithms (Smith, 1996). Two sets of experiments were performed on random problems. The first set compared three methods of backjumping on *CSPs* with 15 variables ($n = 15$). The second set compared the three backjumping algorithms,combined with forward-checking. Here larger problems can be solved and the *CSPs* included 20 variables ($n = 20$). In all of our experiments the number of values for each variable was 10 ($k = 10$). Two values of constraints density were used, $p_1 = 0.3$ and $p_1 = 0.7$. The tightness value p_2, was varied between 0.1 and 0.9, in order to cover all ranges of problem difficulty. For each of the pairs of fixed density and tightness (p_1, p_2), 50 different random problems were solved by each algorithm and the results presented are an average of these 50 runs.

Three backjumping algorithms were compared, Conflict Based Backjumping (CBJ), Dynamic Backtracking (DBT) and Retroactive Dynamic Backtracking (*Retro* DBT). In all of our experiments all the algorithms use a *Min-Domain* heuristic for choosing the next variable to be assigned.

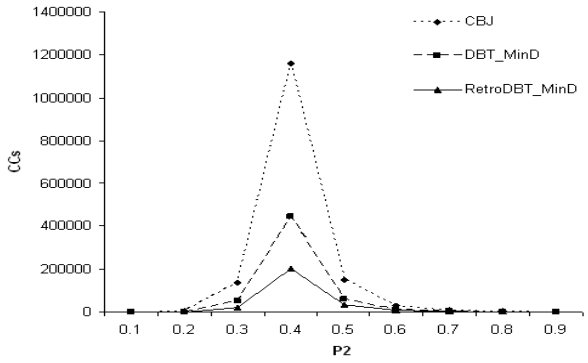

Figure 8: Constraints checks performed by *DBT*, *CBJ* and *Retroactive DBT* ($p_1 = 0.7$).

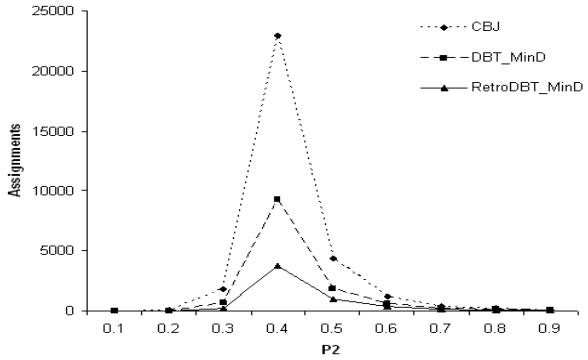

Figure 9: Assignments performed by *DBT*, *CBJ* and *Retroactive DBT* ($p_1 = 0.7$).

Figure 6 presents the number of constraints checks performed by the three algorithms on low density CSPs ($p_1 = 0.3$). The CBJ algorithm does not benefit from the heuristic when it is not combined with forward-checking. The advantage of both versions of DBT over CBJ is therefore large. *Retroactive DBT* improves on standard *DBT* by a large factor as well. Figure 7 presents similar results for the number of assignments performed by the algorithms.

Figures 8 and 9 present the results for high density CSPs ($p_1 = 0.7$). Although the results are similar, the differences between the algorithms are smaller for the case of high density CSPs.

In our second set of experiments, each algorithm was combined with *Forward-Checking* (Prosser, 1993). This improvement enabled the testing of the algorithms on larger CSPs, with 20 variables.

Figure 10 presents the number of constraints checks performed by each of the algorithms. It is very clear that the combination of *CBJ* with forward-checking improves the algorithm and makes it compatible with the others. This is easy to explain since the pruned domains as a result of forward-checking enable an effective use of the Min-Domain heuristic. Both *FC_CBJ* and *Retroactive FC_DBT* outperform *FC_DBT*. *Retroactive FC_DBT* performs better than *FC_CBJ*. Similar results in the number of assignment attempts are presented in Figure 11.

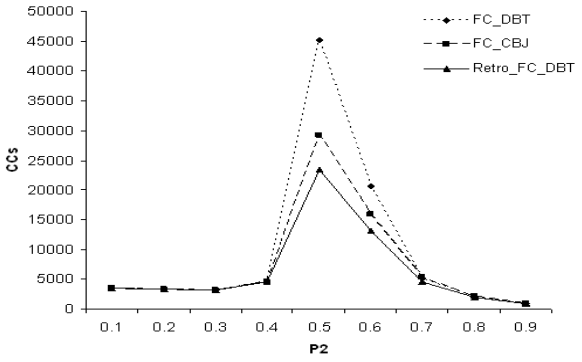

Figure 10: Constraints checks performed by *FC_DBT*, *FC_CBJ* and *FC_Retroactive DBT* ($p_1 = 0.3$).

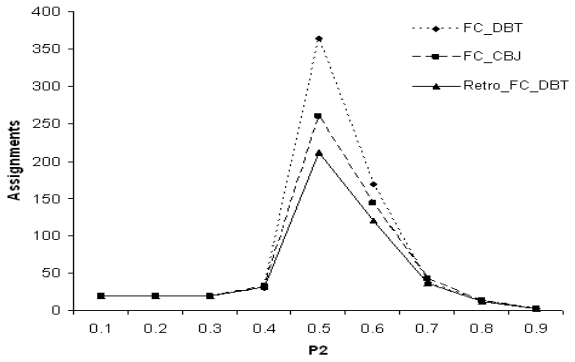

Figure 11: Assignments performed by *FC_DBT*, *FC_CBJ* and *FC_Retroactive DBT* ($p_1 = 0.3$).

Figures 12 and 13 present similar results for high density *CSPs*. As before, the differences between the algorithms are smaller when solving CSPs with higher densities.

6.2 Experiments on Meeting Scheduling Problems (MSPs)

Meeting scheduling is a well-known, recurrent and easily described problem. The meeting scheduling problem (MSP) will be described below as a centralistic constraints satisfaction problem (CSP). However, one of its most interesting features is the fact that it is a Distributed CSP. Informally, a set of agents want to meet. They search for a feasible meeting time that satisfies the private constraints of each of the agents and in addition satisfies arrival-time constraints (among different meetings of the same agent).

The general definition of the MSP family is as follows:

- A group S of m agents

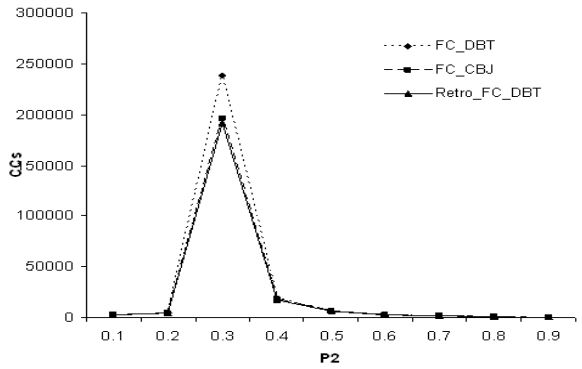

Figure 12: Constraints checks performed by *FC_DBT*, *FC_CBJ* and *FC_Retroactive DBT* ($p_1 = 0.7$).

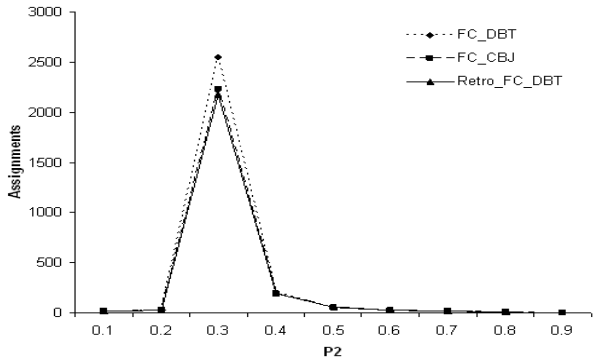

Figure 13: Assignments performed by *FC_DBT*, *FC_CBJ* and *FC_Retroactive DBT* ($p_1 = 0.7$).

- A set T of n meetings
- The duration of each meeting m_i is $duration_i$
- Each meeting m_i is associated with a set s_i of agents in S, that attend it
- Consequently, each agent has a set of meetings that it must attend
- Each meeting is associated with a location
- The scheduled time-slots for meetings in T must enable the participating agents to travel among their meetings

The table below presents an example of a MSP, including the traveling time in time-units (say, hours) between different meeting locations.

Meeting	Location	Attending agents
m_1	L_1	A_1,A_3
m_2	L_2	A_2,A_3,A_4
m_3	L_3	A_1,A_4
m_4	L_4	A_1,A_2

The distances (in time-slots) between the meetings are described in the figure 14.

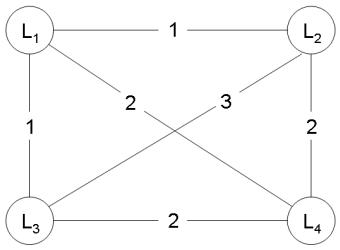

Figure 14: Example of a meeting scheduling problem.

The meeting scheduling problem as described above can be naturally represented as a constraints satisfaction problem (CSP) in the following way:

- A set of variables T - m_1,m_2...,m_n - the meetings to be scheduled
- Domains of values D - all weekly time-slots
- A set of constraints C - for every pair of meetings m_i,m_j there is an arrival-time constraint, if there is an agent that participates in both meetings. Private meetings are equivalent to unary constraints that remove values from domains of some meetings. Since all agents have the same arrival-times between any two locations, there is only one type of arrival-time binary constraint.

arrival-time constraint (AC) - Given two time-slots t_i, t_j there is a conflict if $|time(t_i) - time(t_j)| - duration_i < TravellingTime(location(m_i), location(m_j))$

Simplifying assumptions:

1. All agents have the same size of weekly calendar - M time-slots
2. All Meetings have the same duration 1 time-slot.
3. Each agent attends the same number of meetings

The Density of the *CSP* network depends on the number of meetings (m), the number of agents (n) and the number of meetings per agent (k). The Tightness of a constraint depends on the domain size of the meetings and the locations of the two constrained meetings. The Density and Tightness can be calculated in the following way:

Density (p_1) - the ratio of the total number of edges to the maximal number of possible edges. $p_1 = edgesinthenetwork/(m * (m-1)/2)$

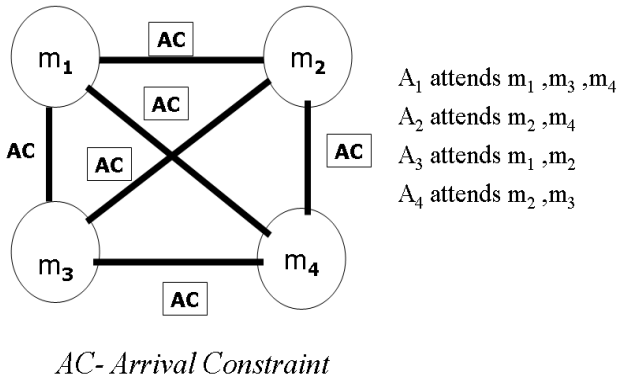

Figure 15: representing a MSP as a CSP.

Tightness (p_2) - the ratio between the total number of eliminated time slots to the total number of tuples (D^2). Therefore p_2 is defined as follows:
$p_2 = (D * (2 * s + 1) - s^2)/(D * D)$, where s is the traveling time between the meetings locations. A representation of a Meeting Scheduling Problems as *CSP* is described in Figure 15

Random Meeting Scheduling Problem (RMSP) specification: The RMSP can be parametrized in many ways. Parameters can be the number of meetings, locations, number of agents, etc. Following is the list of all parameters

- number of meetings - m
- number of agents - n
- number of meetings per agent- k
- distances between locations of meetings - in units of time slots
- domain size - number of time-slots- l

The meetings are the set of m variables of the constraints network, each representing a meeting at a specific location. The domains of values are the time-slots l. An edge between any pair of variables represents an agent that participates in both meetings. The density of the constraints network is a function of the number of edges in the network. The number of edges in the network depends on the number of agents and the distribution of meetings that each agent attends. If each agent participates in k meetings, we generate the resulting *CSP* as follows. For each of the n agents a clique of k variables (meetings) is selected randomly, such that not all of the edges of the clique are already in the network. All the edges of the generated clique are added to the *CSP* network, representing the arrival-time constraints between the meetings of each agent. The arrival-time between each two meetings is also randomly generated. Note, that an agent A_i adds an arrival-constraint between meetings m_j ,m_k only if there is no other agent that attends both meetings. Two agents or more that participate in m_j, m_k define only one arrival-constraint. The distance between locations of meetings randomly generated according to the given range (between the minimal meeting distance and the maximal one).

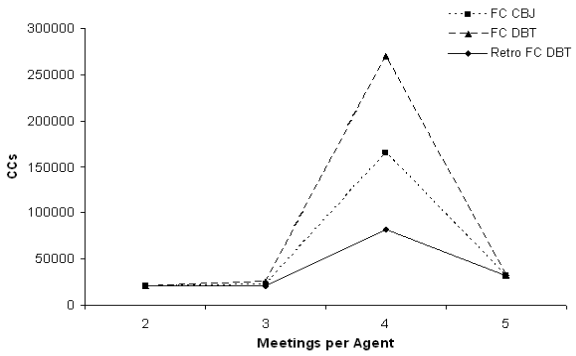

Figure 16: CCs performed by *FC_DBT*, *FC_CBJ* and *Retroactive FC_DBT* solving sparse MSPs (13 agents).

The randomly generated meeting scheduling problems (RMSPs) included 40 meetings and a domain size of 12. The distance between every two meetings was randomly selected between 2, 3 and 4. We evaluated sparse problems with 13 agents and dense problems with 17 agents. The number of meetings per agent was varied between 3 and 5 to evaluate a range of problem difficulty. Figure 16 presents the number of constraints checks performed by the three algorithms while solving random meeting scheduling problems with 13 agents (i.e. sparse problems). *FC_CBJ* performs approximately half the number of constraints checks performed by *FC_DBT*. *Retroactive FC_DBT* improves the results of *FC_CBJ* by a factor larger than 2 and the results of standard DBT by a factor of 6.

Figure 17 presents similar results for the number of assignments performed by the algorithms. A large advantage of *Retroactive FC_DBT* over the other algorithms. On the hardest instances, 3 and 4 meetings per agent, the improvement over standard *FC_DBT* and *FC_CBJ* is by an order of magnitude.

Figures 18 and 19 present similar results for RMSPs with higher density. The improvement factor in these experiments is much larger. In fact, it is hard to get a good perspective on the difference between *FC_CBJ* and *Retroactive FC_DBT* since both of them improve the performance of standard *FC_DBT* by orders of magnitude. Therefore, figures 20 and 21 present a closer look into the performance on these two algorithms on dense MSPs. It is clear that *Retroactive FC_DBT* is much better than *FC_CBJ* for dense RMSPs.

7. Discussion

Variable ordering heuristics such as *Min-Domain* are known to improve the performance of CSP algorithms (Haralick & Elliott, 1980; Bessiere & Regin, 1996; Dechter & Frost, 2002). This improvement results from a reduction in the search space explored by the algorithm. Previous studies have shown that *DBT* does not preserve the properties of variable ordering heuristics since it dynamically places variables during backtracking in a different position than the original position which was selected by the heuristic. As a result, *DBT* was found to perform poorly compared to

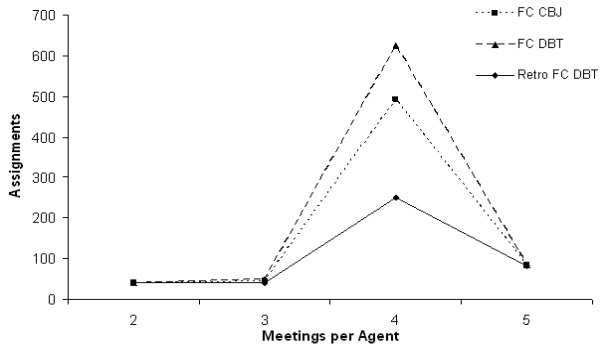

Figure 17: Assignments performed by *FC_DBT*, *FC_CBJ* and *Retroactive FC_DBT* solving sparse MSPs (13 agents).

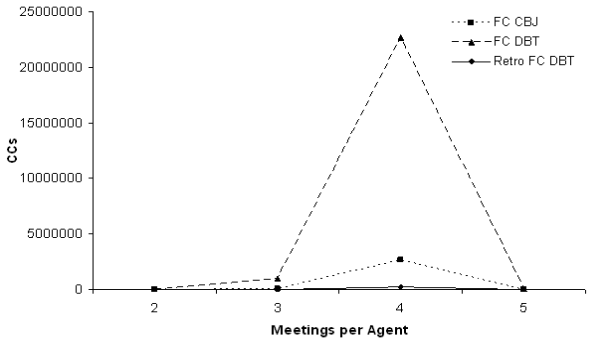

Figure 18: CCs performed by *FC_DBT*, *FC_CBJ* and *Retroactive FC_DBT* solving Dense MSPs (17 agents).

CBJ (Baker, 1994). The *Retroactive DBT* algorithm, presented in this paper, combines the advantages of both previous algorithms by preventing the placing of variables in a position which does not support the heuristic. More importantly, *Retroactive FC_DBT* enables the reordering (or reassigning) of assigned variables that are re-placed after the culprit assignment, after a backtrack operation.

We have used the mechanism of *Dynamic Backtracking* which by maintaining eliminating $Nogoods$, allows variables with higher priority to be reassigned while lower priority variables keep their assignment. These dynamically maintained domains enable to take the *Min-Domain* heuristic to a new level. Standard backtracking algorithms use ordering heuristics only to decide on which variable is to be assigned next. *Retroactive DBT* enables the use of heuristics which reorder assigned variables. Since the sizes of the current domains of variables are dynamic during search, the

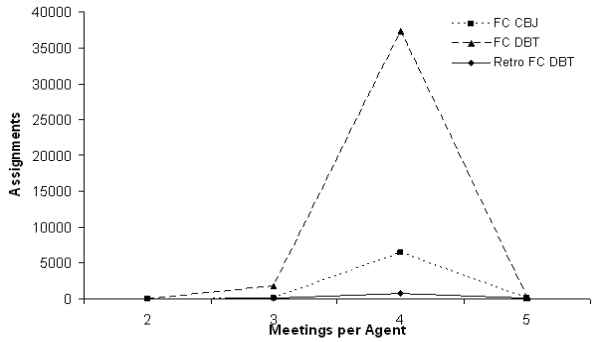

Figure 19: Assignments performed by *FC_DBT*, *FC_CBJ* and *Retroactive FC_DBT* Dense MSPs (17 agents).

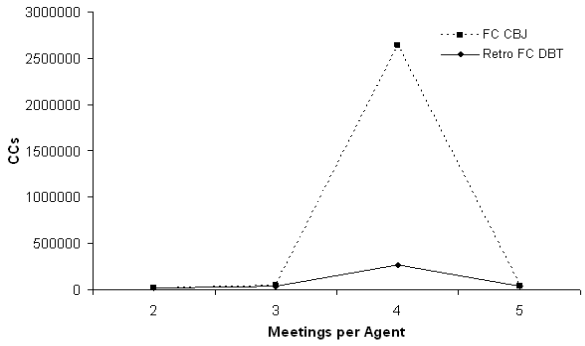

Figure 20: CCs performed by *FC_CBJ* and *Retroactive FC_DBT* solving Dense MSPs (17 agents).

flexibility of the heuristics which are possible in *Retroactive DBT* enables a dynamic enforcement of the *Min-Domain* property over assigned and unassigned variables.

It is especially interesting to see the improved performance of the retroactive ordering approach in the presence of forward-checking. The heuristic idea of selecting *Min-Domain*, performs well for the combination of domain filtering algorithms- *FC* and *DBT*. Both maintaining dynamic domains for all variables during search.

The ordering of assigned variables requires some overhead in computation when the algorithm maintains consistency by using *Forward Checking*. This overhead was found by the experiments to be worth the effort since the overall computation effort is reduced.

Our choice for combining lookahead and backjumping uses forward-checking. Although deeper lookahead methods such as maintaining arc-consistency (*MAC*) were suggested for *DBT* (Jussien & Lhomme, 2002), forward-checking preferred for two reasons. First its simplicity. The combination of standard DBT with MAC results in a very complicated algorithm which its consistency is delicate.

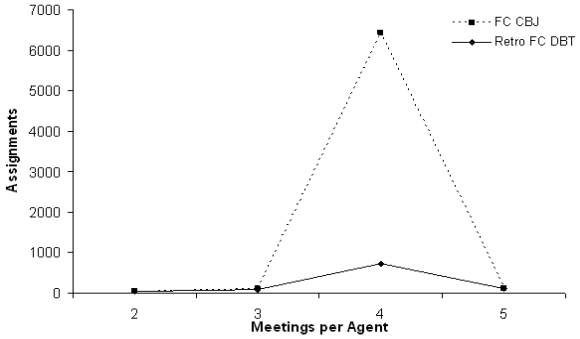

Figure 21: Assignments performed by *FC_CBJ* and *Retroactive FC_DBT* Dense MSPs (17 agents).

Second, a recent study on the properties of ordering heuristics on CSP search algorithms (Wallace, 2005) have shown that the effect of ordering heuristics on *FC* and {*MAC* is similar.

8. Conclusions

Standard ordering heuristics are commonly used in CSP solving algorithms. The standard use of an ordering heuristic is taking an intelligent choice when selecting the next variable to be ordered. In this paper we proposed a new algorithm which takes into consideration the dynamic nature of a CSP solving algorithm takes into an extreme level the exploitation of the current structure of the problem by the heuristic. This is done by enforcing the ordering on assigned variables according to there dynamic state. Our experimental study shows clearly the advantage of the proposed algorithm over the standard heuristics. While on random problems, the factor of improvement is small, on a realistic structured problem the improvement is by a large factor.

References

Baker, A. (1994). The Hazards of Fancy Backtracking. In *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI '94), Volume 1*, pp. 288–293 Seattle, WA, USA. AAAI Press.

Bessiere, C., & Regin, J. (1996). MAC and combined heuristics: two reasons to forsake FC (and CBJ?) on hard problems. In *Proc. CP 96*, pp. 61–75 Cambridge MA.

Chen, X., & van Beek, P. (2001). Conflict-directed backjumping revisited. *Journal of Artificial Intelligence Research (JAIR)*, *14*, 53–81.

Dechter, R., & Frost, D. (2002). Backjump-based backtracking for constraint satisfaction problems. *Artificial Intelligence*, *136:2*, 147–188.

Dechter, R. (2003). *Constraint Processing*. Morgan Kaufman.

Gent, I., & Walsh, T. (1999). CSPLib: a benchmark library for constraints. Tech. rep., Technical report APES-09-1999. Available from http://csplib.cs.strath.ac.uk/. A shorter version appears in

the Proceedings of the 5th International Conference on Principles and Practices of Constraint Programming (CP-99).

Ginsberg, M. L. (1993). Dynamic Backtracking. *J. of Artificial Intelligence Research*, *1*, 25–46.

Haralick, R. M., & Elliott, G. L. (1980). Increasing Tree Search Efficiency for Constraint Satisfaction Problems. *Artificial Intelligence*, *14*, 263–313.

Jussien, N., & Lhomme, O. (2002). Local search with constraint propagation and conflict-based heuristics. *Artificial Intelligence*, *139*(1), 21–45.

Kondrak, G., & van Beek, P. (1997). A Theoretical Evaluation of Selected Backtracking Algorithms. *Artificial Intelligence*, *21*, 365–387.

Maheswaran, R. T., Pearce, J. P., Bowring, E., Varakantham, P., & Tambe, M. (2006). Privacy Loss in Distributed Constraint Reasoning: A Quantitative Framework for Analysis and its Applications. *Autonomous Agents and Multi-Agent Systems*, *13(1)*, 27–60.

Modi, J., & Veloso, M. (2004). Multiagent Meeting Scheduling with Rescheduling. In *Proc. 5th workshop on distributed constraints reasoning DCR-04* Toronto.

Prosser, P. (1993). Hybrid Algorithms for the Constraint Satisfaction Problem. *Computational Intelligence*, *9*, 268–299.

Prosser, P. (1996). An Empirical Study of Phase Transitions in Binary Constraint Satisfaction Problems. *Artificial Intelligence*, *81*, 81–109.

Smith, B. M. (1996). Locating the phase transition in binary constraint satisfaction problems. *Artificial Intelligence*, *81*, 155 – 181.

Wallace, R. J. (2005). Factor Analytic Studies of CSP Heuristics. In *CP-2005*, pp. 712–726 Sigtes (Barcelona), Spain.

Wallace, R. J., & Freuder, E. (2002). Constraint-based multi-agent meeting scheduling: effects of agent heterogeneity on performance and privacy loss. In *Proc. 3rd workshop on distributed constrait reasoning, DCR-02*, pp. 176–182 Bologna.