Conflict directed Backjumping for Max-CSPs*

Roie Zivan and Amnon Meisels
Department of Computer Science,
Ben-Gurion University of the Negev,
Beer-Sheva, 84-105, Israel
{zivanr,am}@cs.bgu.ac.il

Abstract

Constraint Optimization problems are commonly solved using a Branch and Bound algorithm enhanced by consistency maintenance procedures (Wallace and Freuder 1993; Larrosa and Meseguer 1996; Larrosa *et al.* 1999; Larrosa and Schiex 2003; 2004). All these algorithms traverse the search space in a chronological order and gain their efficiency from the quality of the consistency maintenance procedure.

The present study introduces Conflict-directed Backjumping (CBJ) for Branch and Bound algorithms. The proposed algorithm maintains *Conflict Sets* which include only assignments whose replacement can lead to a better solution. The algorithm backtracks according to these sets. CBJ can be added to all classes of the Branch and Bound algorithm. In particular to versions of Branch & Bound that use advanced maintenance procedures of soft local consistency levels, $NC*$, $AC*$ and $FDAC$ (Larrosa and Schiex 2003; 2004). The experimental evaluation of *B&B_CBJ* on random *Max-CSPs* shows that the performance of all algorithms is improved both in the number of assignments and in the time for completion.

Introduction

In standard CSPs, when the algorithm detects that a solution to a given problem does not exist, the algorithm reports it and the search is terminated. In many cases, although a solution does not exist we wish to produce the best complete assignment, i.e. the assignment to the problem which includes the smallest number of conflicts. Such problems from the scope of Max-Constraint Satisfaction Problems (*Max-CSPs*) (Larrosa and Meseguer 1996). Max-CSPs are a special case of the more general Weighted Constraint Satisfaction Problem (WCSPs) (Larrosa and Schiex 2004) in which each constraint is assigned a weight which defines its cost if it is violated by a solution. The cost of a solution is the sum of the weights of all constraints violated by the solution (in Max-CSPs all weights are equal to 1). The requirement in solving WCSPs is to find the minimal cost (optimal) solution. *WCSPs* and *Max-CSPs* are therefore termed *Constraint Optimization Problems*.

*The research was supported by the Lynn and William Frankel Center for Computer Science, and by the Paul Ivanier Center for Robotics.

In this paper we focus for simplicity on *Max-CSP* problems. *Max-CSP* is an optimization problem with a search tree of bounded depth. Like other such optimization problems, the choice for solving it is to use a *Branch and Bound* algorithm (Dechter 2003). In the last decade, various algorithms were developed for Max and Weighted CSPs (Wallace and Freuder 1993; Larrosa and Meseguer 1996; Larrosa *et al.* 1999; Larrosa and Schiex 2004). All of these algorithms are based on standard backtracking and gain their efficiency from the quality of the consistency maintenance procedure they use. In (Larrosa and Schiex 2004), the authors present maintenance procedures of soft local consistency levels, *NC** and *AC**, which improve on former versions of *Node-consistency* and *Arc-consistency*. An improved result for *Max-CSPs* was presented in (Larrosa and Schiex 2003). This result was achieved by enforcing extended consistency which generates higher lower bounds.

The present paper improves on previous results by adding *Conflict-directed Backjumping* to the *B&B* algorithms presented in (Larrosa and Schiex 2004) and in (Larrosa and Schiex 2003). Conflict-directed Backjumping (*CBJ*) is a method which is known to improve standard *CSP* algorithms (Dechter 2003; Ginsberg 1993; Kondrak and van Beek 1997; Zivan and Meisels 2003). In order to perform *CBJ*, the algorithm stores for each variable the set of assignments which caused the removal of values from its domain. When a domain empties, the algorithm backtracks to the last assignment in the corresponding conflict set.

One previous attempt at conflict directed $B\&B$ used reasoning about conflicts during each forward search step (Li and Williams 2005). Conflicts were used in order to guide the forward search away from infeasible and sub-optimal states. This is in contrast to the use of conflict reasonong for backjumping proposed in the present paper. Another attempt to combine conflict directed (intelligent) backtracking with $B\&B$ was reported for the Open-Shop problem (Guéret *et al.* 2000). The algorithm proposed in (Guéret *et al.* 2000) is specific to the problem. Similarly to CBJ for standard $CSPs$, explanations for the removal of values from domains are recorded and used for resolving intelligently the backtrack destination.

Performing back-jumping for *Max-CSPs* is more complicated than for standard *CSPs*. In order to generate a consistent conflict set, all conflicts that have contributed to the

current lower bound must be taken into consideration. Furthermore, additional conflicts with unassigned values with equal or higher costs must be added to the conflict set in order to achieve completeness.

The required information needed for the detection of the culprit variables that will be the targets for the algorithm backjumps is polynomial and the maintenance of the data structures does not require additional iterations of the algorithm.

The results presented in this paper show that *CBJ* decreases the number of assignments performed by the algorithm by a large factor. The improvement in time is dependent on the degree of the consistency maintenance procedure used.

Max-CSPs are presented in Section . A description of the standard *Branch and Bound* algorithm along with the maintenance procedures of $NC*$, $AC*$ and $FDAC$ is presented in Section . Section presents the *CBJ* algorithm for *Branch and Bound* with $NC*$, $AC*$ and $FDAC$. A correctness proof for *B&B_CBJ* is presented in Section . An extensive experimental evaluation, of the contribution of conflict directed backjumping to $B\&B$ with $NC*$, $AC*$ and $FDAC$, is presented in Section . The experiments were conducted on randomly generated *Max-CSPs*.

Max Constraint Satisfaction Problems

A *Max - Constraint Satisfaction Problem* (*Max-CSP*) is composed, like standard *CSP*, of a set of n variables $X_1, X_2, ..., X_n$. 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 $X_{i_k}, X_{j_l}, ..., X_{m_n}$, with domains of values for each variable $D_{i_k}, D_{j_l}, ..., D_{m_n}$, the constraint is defined as $R \subseteq D_{i_k} \times D_{j_l} \times ... \times D_{m_n}$. Similarly to former studies of *Max-CSPs* (Larrosa and Meseguer 1996; Larrosa *et al.* 1999; Larrosa and Schiex 2003), we assume that all constraints are binary. A binary constraint R_{ij} between any two variables X_j and X_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 set of assignments of values to a set of variables. The $cost$ of a partial solution in a *Max-CSP* is the number of conflicts violated by it. An optimal **solution** to a *Max-CSP* is a partial solution that includes all variables and which includes a minimal number of unsatisfied constraints, i.e. a solution with a minimal cost.

The Branch and Bound algorithm

Optimization problems with a finite search-space are often solved by a Branch and Bound ($B\&B$) algorithm. Both *Weighted CSPs* and *Max-CSPs* fall into this category. The overall framework of a $B\&B$ algorithm is rather simple. Two bounds are constantly maintained by the algorithm, an $upper\_bound$ and a $lower\_bound$. The $upper\_bound$ holds the best solution found so far and the $lower\_bound$ holds the cost of the *current partial solution*, which the algorithm is currently trying to expand. When the $lower\_bound$ is equal to or higher than the $upper\_bound$, the algorithm attempts to replace the most recent assignment. If all values of a variable fail, the algorithm backtracks to the most recent variable assigned (Wallace and Freuder 1993; Larrosa and Schiex 2004).

```
B&B
1.  current_state ← initial_state;
2.  CPS ← empty_assignment;
3.  i ← 0;
4.  while(i ≥ 0)
5.      if(i = n)
6.          upper_bound ← lower_bound;
7.          i ← i − 1;
8.      else foreach (a ∈ D_i)
9.          temp_state ← update_state(i, a);
10.         if(local_consistent(temp_state))
11.             states[i] ← current_state;
12.             current_state ← temp_state;
13.             i ← i + 1;
14.     i ← find_culprit_variable();
15.     current_state ← states[i];

update_state(i, val)
1.  add (i, val) to CPS;
2.  CPS.cost ← CPS.cost + cost(i, val);
3.  for j ← i + 1 to n − 1
4.      foreach (a ∈ D_j)
5.      if(conflicts(⟨i, val⟩, ⟨j, a⟩))
6.          cost(a, i) ← cost(a, i) + 1;
```

Figure 1: Standard B&B algorithm

The *Branch and Bound* algorithm is presented in Figure 1. The general structure of the algorithm is different than its recursive presentation by Larrosa and Schiex in (Larrosa and Schiex 2004). In order to be able to perform backjumping, we need an iterative formulation (cf. (Prosser 1993)). We use an array of $states$ which holds for each successful assignment the state of the algorithm before it was performed. After each backtrack the current state of the algorithm will be set to the state which was stored before the culprit assignment was performed. The space complexity stays the same as in the recursive procedure case, i.e. larger than a single state by a factor of n.

We assume all costs of all values are initialized to zero. The index of the variable which the algorithm is currently trying to assign is i. It is initialized to the index of the first variable, 0 (line 3). In each iteration of the algorithm, an attempt to assign the current variable i is performed. The assignment attempt includes the updating of the current state of the algorithm according to the selected assigned value and checking if the new state is consistent (lines 8-10). For standard *B&B* algorithm the $lower\_bound$ is equal to the cost of the current partial solution (CPS). thus the consistency check is simply a verification that the cost of the CPS is smaller than the $upper\_bound$. If the state is consistent, an update of the current state is performed and the current index i is incremented (lines 11-13). When all the values of a variable are exhausted the algorithm backtracks (lines 14-15). In a version with no *backjumping*, the function **find_culprit_variable** simply returns $i - 1$. The algorithm terminates when the value of the index i is lower than the

index of the first variable i.e. smaller than zero (line 4).

Procedure **update_state** is also presented in Figure 1. lines 1,2 update the CPS and its cost with the new assignemnt. Then, the cost of each value of an unassigned variable which is in conflict with the new assignment is incremented (lines 3-6).

The naive and exhaustive $B\&B$ algorithm can be improved by using *consistency check* functions which increase the value of the $lower\_bound$ of a *current partial solution*. After each assignment, the algorithm performs a consistency maintenance procedure that updates the costs of potential future assignments and increases its chance to detect early a need to backtrack. Three of the most successful *consistency check* functions are described next.

Node Consistency and NC*

Node Consistency (analogous to Forward-checking in standard CSPs) is a very standard consistency maintenance method in standard $CSPs$ (Dechter 2003). The main idea is to ensure that in the domains of each of the unassigned variables there is at least one value which is consistent with the current partial solution. In standard $CSPs$ this would mean that a value has no conflicts with the assignments in the *current partial solution*. In *Max-CSPs*, for each value in a domain of an unassigned variable, one must determine if assigning it will increase the $lower\_bound$ beyond the limit of the $upper\_bound$. To this end, the algorithm maintains for every value a $cost$ which is its number of conflicts with assignments in the *current partial solution*. After each assignment, the costs of all values in domains of unassigned variables are updated. When the sum of a value's cost and the cost of the *current partial solution* is higher or equal to the $upper\_bound$, the value is eliminated from the variable's domain. An empty domain triggers a backtrack.

The down side of this method in *Max-CSPs* is that the number of conflicts counted and stored at the value's $cost$, does not contribute to the global $lower\_bound$, and it affects the search only if it exceeds the $upper\_bound$. In (Larrosa and Schiex 2004), the authors suggest an improved version of Node Consistency they term *NC**. In *NC** the algorithm maintains a global cost C_ϕ which is initially zero. After every assignment, all costs of all values are updated as in standard NC. Then, for each variable, the minimal cost of all values in its domain, c_i, is added to C_ϕ and all value costs are decreased by c_i. This means that after the method is completed in every step, the domain of every unassigned variable includes one value whose cost is zero. The global $lower\_bound$ is calculated as the sum of the *current partial solution*'s cost and C_ϕ.

Figure 2 presents an example of the operation of the $NC*$ procedure on a single variable. On the left hand side, the values of the variable are presented with their cost before the run of the $NC*$ procedure. The value of the global cost C_ϕ is 6. The minimal cost of the values is 2. On the RHS, the state of the variable is presented after the $NC*$ procedure. All costs were decreased by 2 and the global value C_ϕ is raised by 2.

Any value whose $lower\_bound$, i.e. the sum of the *current partial solution*'s cost, C_ϕ and its own cost, exceeds the

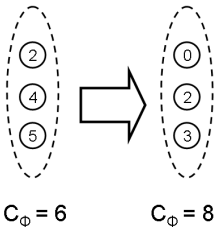

Figure 2: Values of a variable before and after running NC*

NC*(i**)**
1. **for** $j \leftarrow (i+1)$ to $(n-1)$
2. $\quad c_j \leftarrow min\_cost(D_j)$;
3. $\quad$ **foreach** $(a \in D_j)$
4. $\quad\quad a.cost \leftarrow a.cost - c_j$;
5. $\quad C_\phi \leftarrow C_\phi + c_j$;
6. $lower\_bound \leftarrow distance.cost + C_\phi$;
7. **return** $(lower\_bound < upper\_bound)$;

Figure 3: Standard B&B algorithm

limit of the $upper\_bound$, is removed from the variable's domain as in standard NC (Larrosa and Schiex 2004).

The $NC*$ consistency maintenance function is presented in Figure 3. The function finds for each variable the minimal cost among its values and decreases that cost from all the values in the domain (lines 1-4). Then it adds the minimal cost to the global cost C_ϕ (line 5). After the new cost of C_ϕ is calculated it is added to the distance cost in order to calculate the current $lower\_bound$ (line 6). The function returns *true* if the calculated $lower\_bound$ is smaller than the $upper\_bound$ and *false* otherwise.

Arc Consistency and AC*

Another consistency maintenance procedure which is known to be effective for $CSPs$ is *Arc Consistency*. The idea of standard AC (Bessiere and Regin 1995) is that if a value v of some unassigned variable X_i, is in conflict with all values in the current domain of another unassigned variable X_j then v can be removed from the domain of X_i since assigning it to X_i will cause a conflict.

In *Max-CSPs*, a form of Arc-Consistency is used to project costs of conflicts between unassigned variables (Larrosa and Meseguer 1996; Larrosa *et al.* 1999; Larrosa and Schiex 2003; 2004). As for standard $CSPs$, a value in a domain of an unassigned variable, which is in conflict with all the values in the current domain of another unassigned variable, will cause a conflict when it is assigned. This information is used in order to project the cost from the binary constraint to the cost associated with each value. Values for which the sum of their cost and the global $lower\_bound$ exceeds the $upper\_bound$, are removed from the domain of their variable (Larrosa and Schiex 2004). $AC*$ combines the advantages of AC and $NC*$. After performing AC, the updated cost of the values are used by the $NC*$ procedure to increase the global cost C_ϕ. Values are removed as in $NC*$ and their removal initiates the rechecking of AC. The system is said to be $AC*$ if it is both AC and $NC*$ (i.e. each

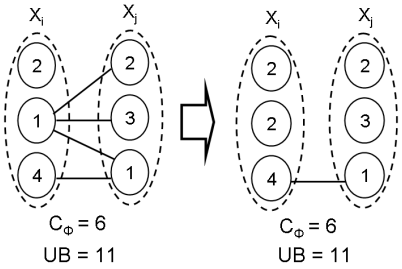

Figure 4: performing AC* on two unassigned variables

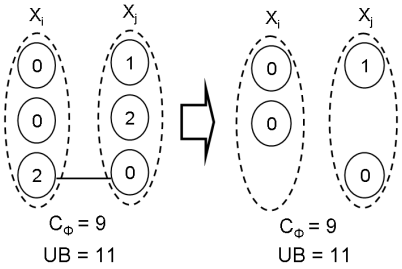

Figure 5: performing AC* on two unassigned variables

domain has a value with a zero cost) (Larrosa and Schiex 2004).

Figures 4 and 5 present an example of the AC* procedure. On the LHS of Figure 4 the state of two unassigned variables, X_i and X_j is presented. The center value of variable X_i is constrained with all the values of variable X_j. Taking these constraints into account, the cost of the value is incremented and the result is presented on the RHS of Figure 4. The left hand side of Figure 5 presents the state after the process of adding the minimum value cost to C_ϕ and decreasing the costs of values of both X_i and X_j. Since the minimal value of X_i was 2 and of X_j was 1, C_ϕ was incremented by 3. Values for which the sum of C_ϕ and their cost is equal to the $upper\_bound$ are removed from their domains and the procedure ends with the state on the RHS of Figure 5.

Full Directed Arc Consistency

The $FDAC$ consistency method enforces a stronger version of Arc-Consistency than $AC*$ (cf. (Larrosa and Schiex 2003)). Consider a CSP with an order on its unassigned variables. If for each value val_{i_k} of variable V_i, in every domain of an unassigned variable V_j which is placed after V_i in the order, a value val_{j_s} has a cost of zero and there is no binary constraint between val_{i_k} and val_{j_s}, we say that the CSP is in a DAC state. A CSP is in a $FDAC$ state if it is both DAC and $AC*$. [1]

Figure 6 presents on the LHS the domains of two ordered variables X_i and X_j which are not $FDAC$ with respect to this order. On the RHS, an equivalent state of these two variables which is $FDAC$ is presented. This new state was reached by extending the unary constraint of the second

[1]The code for the $AC*$ and $FDAC$ procedures is not used in this paper therefore the reader is referred to (Larrosa and Schiex 2003; 2004).

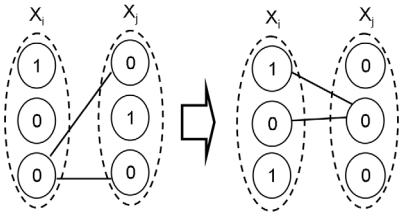

Figure 6: performing FDAC on two unassigned variables

value in X_j's domain to the binary constraint between the two variables. In other words, the cost of this value is decreased by one and a constraint with each of the values in the domain of X_i is added. Then, the binary constraint of the third value of X_i with all the values of X_j is projected to its unary cost.

For a detailed description of $FDAC$ and demonstrations of how $FDAC$ increases the $lower\_bound$ the reader is referred to (Larrosa and Schiex 2003).

Branch and Bound with CBJ

The use of Backjumping in standard *CSP* search is known to improve the run-time performance of the search by a large factor (Prosser 1993; Kondrak and van Beek 1997; Dechter and Frost 2002). Conflict directed Backjumping (*CBJ*) maintains a set of conflicts for each variable, which includes the assignments that caused a removal of a value from the variable's domain. When a backtrack operation is performed, the variable that is selected as the target, is the last variable in the conflict set of the backtracking variable. In order to keep the algorithm complete during backjumping, the conflict set of the target variable, is updated with the union of its conflict set and the conflict set of the backtracking variable (Prosser 1993).

The data structure of conflict sets which was described above for *CBJ* on standard $CSPs$ can be used for the $B\&B$ algorithm for solving *Max-CSPs*. However, additional aspects must be taken into consideration for the case of *Max-CSPs*.

In the description of the creation and maintenance of a consistent conflict set in a $B\&B$ algorithm the following definitions are used:

Definition 1 *A* global $conflict\_set$ *is the set of assignments such that the algorithm back-jumps to the latest assignment of the set.*

Definition 2 *The* $current\_cost$ *of a variable is the cost of its assigned value, in the case of an assigned variable, and the minimal cost of a value in its* $current\_domain$ *in the case of an unassigned variable.*

Definition 3 *A* $conflict\_list$ *of value* v_j *from the domain of variable* X_i*, is the ordered list of assignments of variables in the* $current\_partial\_solution$*, which were assigned before* i*, and which conflict with* v_j*.*

Definition 4 *The* $conflict\_set$ *of variable* X_i *with cost* c_i *is the union of the first (most recent)* c_i *assignments in each of the* $conflict\_lists$ *of all its values (If the* $conflict\_list$ *is shorter than* c_i *then all of it is included in the union).*

$update\_state(i, val)$
1. add (i, val) to CPS;
2. $CPS.cost \leftarrow CPS.cost + cost(i, val)$;
3. **foreach**$(a \in D_i)$
4. **for** 1 to $val.cost$
5. $GCS \cup first\_element$ in $a.conflict\_list$;
6. **remove** $first\_element$ from $a.conflict\_list$
7. **for** $j \leftarrow i + 1$ to $n - 1$
8. **foreach** $(a \in D_j)$
9. **if**(conflicts($\langle i, val\rangle, \langle j, a\rangle$))
10. $a.cost \leftarrow a.cost + 1$;
11. $a.conflict\_list \cup (i, val)$;

find_culprit_variable(i)
1. $culprit \leftarrow$ last assignment in GCS;
2. $GCSleftarrowGCS \setminus culprit$;
3. **return** culprit;

Figure 7: Changes in $B\&B$ required for backjumping

In the case of simple $B\&B$, the *global* $conflict\_set$ is the union of all the $conflict\_sets$ of all assigned variables. Values are always assigned using the min-conflict heuristic i.e. the next value to be assigned is the value with the smallest cost in the variable's current domain. Before assigning a variable the cost of each value is calculated by counting the number of conflicts it has with the $current\_partial\_solution$. Next, the variable's $current\_cost$ is determined to be the lowest cost among the values in its $current\_domain$. As a result, the variable's $conflict\_set$ is generated. The reason for the need to add the conflicts of all values to a variable's $conflict\_set$ and not just the conflicts of the assigned value, is that all the possibilities for decreasing the minimal number of conflicts of any of the variables' values must be explored. Therefore, the latest assignment that can be replaced, and possibly decrease the cost of one of the variables values to be smaller than the variable's current cost must be considered.

Figure 7 presents the changes needed for adding conflict directed backjumping to standard *B&B*. After a successful assignment is added to the distance and its cost is added to the distance cost (lines 1,2), the added cost, $val.cost$ is used to determine which assignments are added to the global conflict set (GCS). For each of the values in the domain of variable i, the first $val.cost$ assignments are removed and added to the GCS (lines 3-6). As a result, when performing a backjump, all that is left to do in order to find the culprit variable is to take the latest assignment (the one with the highest variable index) in the GCS.

The space complexity of the overhead needed to perform *CBJ* in $B\&B$ is simple to bound from above. For a *CSP* with n variables and d values in each domain, the worst case is that for each value the algorithm holds a list of $O(n)$ assignments. This makes the space complexity of the algorithm bounded by $O(n^2d)$.

Figure 8 presents the state of three variables which are included in the *current partial solution*. Variables X_1, X_2 and X_3 were assigned values v_1, v_2 and v_1 respectively. All costs of all values of variable X_3 are 1. The $conflict\_set$ of variable X_3 includes the assignments of X_1 and X_2 even though its assigned value is not in conflict with the assignment of X_2. However, replacing this assignment can lower the cost of value v_2 of variable X_3.

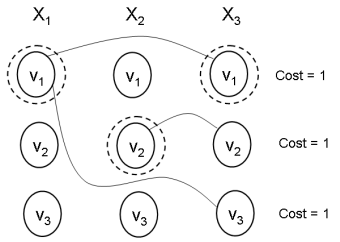

Figure 8: A conflict set of an $assigned$ variable

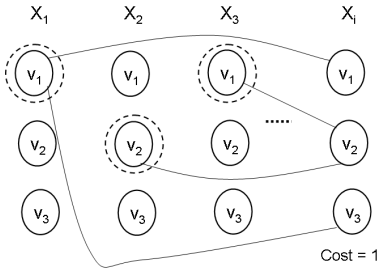

Figure 9: A conflict set of an $unassigned$ variable

Node Consistency with CBJ

In order to perform conflict directed backjumping in a $B\&B$ algorithm that uses node consistency maintenance, the $conflict\_sets$ of unassigned variables must be maintained. To achieve this goal, for every value of a future variable a $conflict\_list$ is initialized and maintained. The $conflict\_list$ includes all the assignments in the *current partial solution* which conflict with the corresponding value. The length of the $conflict\_list$ is equal to the cost of the value. Whenever the $NC*$ procedure adds the cost c_i of the value with minimal cost in the domain of X_i to the global cost C_ϕ, the first c_i assignments in each of the $conflict\_lists$ of X_i's values are added to the *global conflict_set* and removed from the value's $conflict\_lists$. This includes all the values of X_i including the values removed from its domain. Backtracking to the head of their list can cause the return of removed values to the variables $current\_domain$. This means that after each run of the $NC*$ procedure, the *global conflict_set* includes the union of the $conflict\_sets$ of all assigned and unassigned variables.

Figure 9 presents the state of an unassigned variable X_i. The *current partial solution* includes the assignments of three variables as in the example in Figure 8. Values v_1 and v_3 of variable X_i are both in conflict only with the assignment of variable X_1. Value v_2 of X_i is in conflict with the assignments of X_2 and X_3. X_i's cost is 1 since that is the minimal cost of its values. Its conflict set includes the assignments of X_1 since it is the first in the $conflict\_list$ of v_1 and v_3, and X_2 since it is the first in the $conflict\_list$ of v_2. After the $NC*$ procedure, C_ϕ will be incremented by one and the assignments of X_1 and X_2 will be added to the *global conflict_set*.

Figure 10 presents the changes in $NC*$ that are required

NC*(i)
1. **for** $j \leftarrow i+1$ to $n-1$
2. $c_j \leftarrow min\_cost(D_j)$;
3. **foreach** ($a \in D_j$)
4. $a.cost \leftarrow a.cost - c_j$;
5. **for** 1 to c_j
6. $GCS \leftarrow GCS \cup first\_element$ in $a.conflict\_list$;
7. **remove** $first\_element$ from $a.conflict\_list$;
8. $C_\phi \leftarrow C_\phi + c_j$;
9. $lower\_bound \leftarrow CPS.cost + C_\phi$;
10. **return** ($lower\_bound < upper\_bound$);

Figure 10: Changes in $NC*$ maintenance that enable CBJ

for performing CBJ. For each value whose cost is decreased by the minimal cost of the variable, c_j, the first c_j assignments in its $conflict\_list$ are removed and added to the global conflict set (lines 5-7).

AC* and FDAC with CBJ

Adding ***CBJ*** to a $B\&B$ algorithm that includes arc-consistency is very similar to the case of node consistency. Whenever a minimum cost of a future variable is added to the global cost C_ϕ, the prefixes of all of its value's $conflict\_lists$ are added to the *global* $conflict\_set$. However, in $AC*$, costs of values can be incremented by conflicts with other unassigned values. As a result the cost of a variable may be larger than the length of its $conflict\_list$. In order to find the right conflict set in this case one must keep in mind that except for an empty *current partial solution*, a cost of a value v_k of variable X_i is increased due to arc-consistency only if there was a removal of a value which is not in conflict with v_k, in some other unassigned variable X_j. This means that replacing the last assignment in the *current partial solution* would return the value which is not in conflict with v_k, to the domain of X_j. Whenever a cost of a value is raised by arc-consistency, the last assignment in the *current partial solution* must be added to the end of the value's $conflict\_list$. This addition restores the correlation between the length of the $conflict\_list$ and the cost of the value. The variables' $conflict\_set$ and the *global conflict_set* can be generated in the same way as for $NC*$.

Maintaining a consistent $conflict\_set$ in $FDAC$ is somewhat similar to $AC*$. Whenever a cost of a value is extended to a binary constraint, its cost is decreased and its $conflict\_list$ is shortened. When a binary constraint is projected on a value's cost, the cost is increased and the last assignment in the *current partial solution* is added to its $conflict\_list$. Caution must be taken when performing the assignment since the constant change in the cost cost of values may interfere with the min-cost order of selecting values. A simple way to avoid this problem is to maintain for each value a different cost which we term $priority\_cost$. The $priority\_cost$ is updated whenever the value's cost is updated except for updates performed by the DAC procedure. When we choose the next value to be assigned we break ties of costs using the value of the $priority\_cost$.

Correctness of $B\&B\_CBJ$

In order to prove the correctness of the ***B&B_CBJ*** algorithm it is enough to show that the *global conflict_set* maintained by the algorithm is correct. First we prove the correctness for the case of simple $B\&B\_CBJ$ with no consistency maintenance procedure by proving that the replacement of assignments which are not included in the *global conflict_set* cannot lead to lower cost solutions. Consider the case that a *current partial solution* has a length k and the index of the variable of the latest assignment in the *current partial solution*'s corresponding $conflict\_set$ is l. Assume in negation, that there exists an assignment in the *current partial solution* with a variable index $j > l$, that by replacing it the cost of a *current partial solution* of size k with an identical prefix of size $j-1$ can be decreased. Since the assignment j is not included in the *global conflict_set* this means that for every value of variables $X_{j+1}...X_k$, assignment j is not included in the prefix of size $cost$ of any of their values' $conflict\_lists$. Therefore, replacing it would not decrease the cost of any value of variables $X_{j+1}...X_k$ to be lower than their current cost. This contradicts the assumption. □

Next, we prove the consistency of the global conflict set (GCS) in $B\&B\_CBJ$ with the $NC*$ consistency maintenance procedure. To this end we show that all variables in the *current partial solution* whose constraints with unassigned variables contribute to the global cost C_ϕ are included in the *global conflict_set*. The above proof holds for the assignments added due to conflicts within the *current partial solution*. For assignments added to the *global conflict_set* due to conflicts of unassigned variables with assignments in the *current partial solution* we need to show that all conflicting assignments which can reduce the cost of any unassigned variable are included in the *global conflict_set*. After each assignment and run of the $NC*$ procedure, the costs of all unassigned variables are zero. If some assignment of variable X_j in the *current partial solution* was not added to the *global conflict_set* it means that it was not a prefix of any $conflict\_list$ of a size, equal to the cost added to C_ϕ. Consequently, changing an assignment which is not in the *global conflict_set* cannot affect the global $lower\_bound$. □

The consistency of the *global conflict_set* for $AC*$ follows immediately from the correctness of the *conflict_set* for the$B\&B$'s *current partial solution* and for $NC*$. The only difference between $NC*$ and $AC*$ is the addition of the last assignment in the *current partial solution* to the *global conflict_set* for an increment of the cost of some value which was caused by an arc consistency operation. A simple induction proves that at any step of the algorithm, only a removal of a value can cause an increment of a value's cost due to arc consistency. The correctness for the case of $FDAC$ follows immediately from the proof of $AC*$. The details are left out for lack of space.

Experimental Evaluation

The common approach in evaluating the performance of ***CSP*** algorithms is to measure time in logic steps to eliminate implementation and technical parameters from affecting the results. Two measures of performance are used by the present evaluation. The total number of assignments and cpu-time. This is in accordance with former work on ***Max-CSP*** algorithms' performance (Larrosa and Schiex 2003;

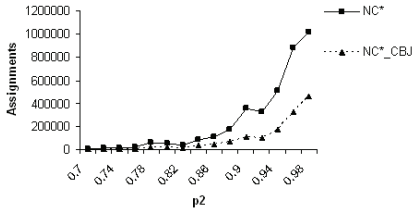

Figure 11: Assignments of *NC** and *NC*_CBJ* (p_1 = 0.4)

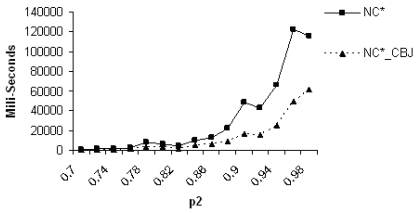

Figure 12: Run-time of *NC** and *NC*_CBJ* (p_1 = 0.4)

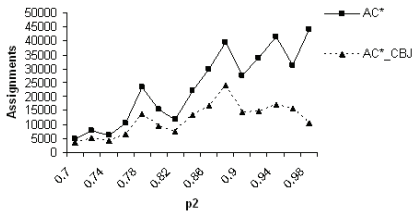

Figure 13: Assignments of *AC** and *AC*_CBJ* (p_1 = 0.4)

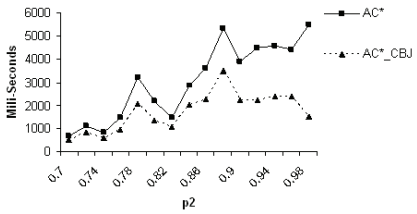

Figure 14: Run-time of *AC** and *AC*_CBJ* (p_1 = 0.4)

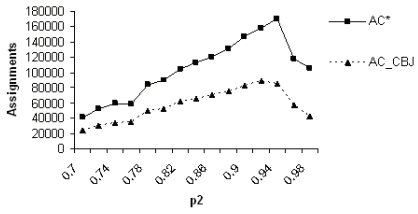

Figure 15: Assignments of *AC** and *AC*_CBJ* (p_1 = 0.9)

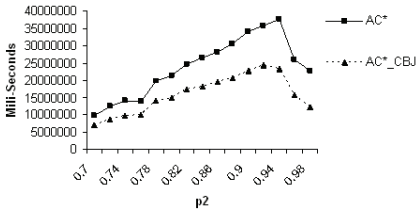

Figure 16: Run-time of *AC** and *AC*_CBJ* (p_1 = 0.9)

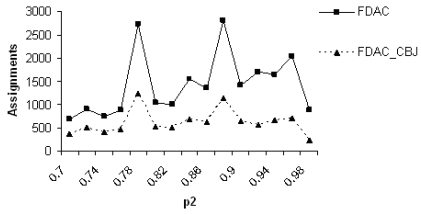

Figure 17: Assignments of *FDAC* and *FDAC_CBJ* (p_1 = 0.4)

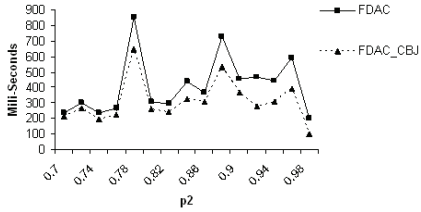

Figure 18: Run-time of *FDAC* and *FDAC_CBJ* (p_1 = 0.4)

2004).

Experiments were conducted on random *CSPs* of n variables, k values in each domain, a constraint density of p_1 and tightness p_2 (which are commonly used in experimental evaluations of *Max-CSP* algorithms (Larrosa and Meseguer 1996; Larrosa *et al.* 1999; Larrosa and Schiex 2004)). In all of the experiments the *Max-CSPs* included 10 variables ($n = 10$) and 10 values for each variable ($k = 10$). Two values of constraint density $p_1 = 0.4$ and $p_1 = 0.9$ were used to generate the *Max-CSPs*. The tightness value p_2, was varied between 0.7 and 0.98, since the hardest instances of *Max-CSPs* are for high p_2 (Larrosa and Meseguer 1996; Larrosa *et al.* 1999). For each pair of fixed density and tightness ($p1$, $p2$), 50 different random problems were solved by each algorithm and the results presented are an average of these 50 runs.

In order to evaluate the contribution of *Conflict directed Backjumping* to *Branch and Bound* algorithms using consistency maintenance procedures, the $B\&B$ algorithm with $NC*$, $AC*$ and $FDAC$ procedures were implemented. The results presented show the performance of these algorithms with and without *CBJ*. The $NC*$ procedure was tested only for low density problems $p_1 = 0.4$, since it does not complete in a reasonable time for $p_1 = 0.9$.

Figure 11 presents the number of assignments performed by *NC** and *NC*_CBJ*. For the hardest instances, where p_2 is higher than 0.9, *NC*_CBJ* outperforms *NC** by a factor of between 3 at $p_2 = 0.92$ and 2 at $p_2 = 0.99$. Figure 12 shows similar results for cpu-time.

Figure 13 presents the number of assignments performed

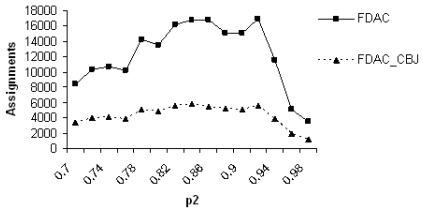

Figure 19: Assignments of *FDAC* and *FDAC_CBJ* ($p_1 = 0.9$)

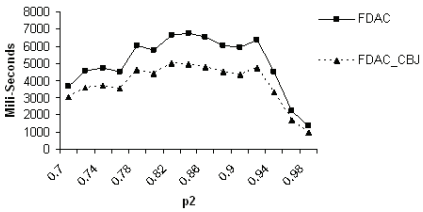

Figure 20: Run-time of *FDAC* and *FDAC_CBJ* ($p_1 = 0.9$)

by *AC** and *AC*_CBJ*. For the hardest instances, where p_2 is higher than 0.9, *AC*_CBJ* outperforms *AC** by a factor of 2. Figure 14 presents the result in cpu-time. The results in cpu-time are similar but the difference is smaller than for the number of assignments.

Figures 15, 16 shows similar results for the *AC** algorithm solving high density *Max-CSPs* ($p1 = 0.9$). The factor of improvement is similar to the low density experiments for both measures.

Figure 17 presents the number of assignments performed by *FDAC* and *FDAC_CBJ*. The difference in the number of assignments between the conflict-directed backjumping version and the standard version is much larger than for the case of *NC** and *AC**. However, the difference in cpu-time, presented in Figure 18, is smaller than for the previous procedures. These differences are also presented for high density *Max-CSPs* in Figures 19 and 20.

The big difference between the results in number of assignments and in cpu-time for deeper look-ahead algorithms can be explained by the large effort that is spent in $FDAC$ for detecting conflicts and pruning during the first steps of the algorithm run. For deeper look-ahead, the backjumping method avoids assignment attempts which require a small amount of computation. The main effort having been made during the assignments of the first variables of the *CSP* which are performed similarly, in both versions.

Conclusions

Branch and Bound is the most common algorithm used for solving optimization problems with a finite search space (such as *Max-CSPs*). Former studies improved the results of standard Branch and Bound algorithms by improving the consistency maintenance procedure they used (Wallace and Freuder 1993; Larrosa and Meseguer 1996; Larrosa *et al.* 1999; Larrosa and Schiex 2003; 2004). In this study we adjusted *CBJ* which is a common technique for standard *CSP* search (Prosser 1993; Kondrak and van Beek 1997) to Branch and Bound with extended consistency maintenance procedures. The results presented in Section show that *CBJ* improves the performance of all versions of the $B\&B$ algorithm. The improvement is measured by two separate measures - the number of assignments performed by the algorithm and its run-time. The factor of improvement in the number of assignments is consistent and large. The factor of improvement in run-time is dependent on the consistency maintenance procedure used. The factor of improvement does not decrease (and even grows) for random problems with higher density.

References

C. Bessiere and J.C. Regin. Using bidirectionality to speed up arc-consistency processing. *Constraint Processing (LNCS 923)*, pages 157–169, 1995.

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

Rina Dechter. *Constraints Processing*. Morgan Kaufman, 2003.

M. L. Ginsberg. Dynamic backtracking. *J. of Artificial Intelligence Research*, 1:25–46, 1993.

Christelle Guéret, Narendra Jussien, and Christian Prins. Using intelligent backtracking to improve branch and bound methods: an application to open-shop problems. *European Journal of Operational Research*, 127(2):344–354, 2000.

G. Kondrak and P. van Beek. A theoretical evaluation of selected backtracking algorithms. *Artificial Intelligence*, 21:365–387, 1997.

J. Larrosa and P. Meseguer. Phase transition in max-csp. In *Proc. ECAI-96*, Budapest, 1996.

J. Larrosa and T. Schiex. In the quest of the best form of local consistency for weighted csp. In *Proc. IJCAI-2003*, Acapulco, 2003.

J. Larrosa and T. Schiex. Solving weighted csp by maintaining arc consistency. *Artificial Intelligence*, 159:1–26, 2004.

J. Larrosa, P. Meseguer, and T. Schiex. Maintaining reversible dac for max-csp. *Artificial Intelligence*, 107:149–163, 1999.

H. Li and B. Williams. Generalized conflict learning for hybrid discrete/linear optimization. In *CP-2005*, pages 415–429, Sigtes (Barcelona), Spain, 2005.

P. Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9:268–299, 1993.

R. J. Wallace and E. C. Freuder. Conjunctive width heuristics for maximal constraint satisfaction. In *Proc. AAAI-93*, pages 762–768, 1993.

R. Zivan and A. Meisels. Synchronous vs asynchronous search on discsps. In *Proc. 1st European Workshop on Multi Agent System, EUMAS*, Oxford, December 2003.