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

Abstract

Max-CSPs are Constraint Optimization Problems that are commonly solved using a Branch and Bound algorithm. The $B\&B$ algorithm was 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 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.

1 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 form the scope of *Max-CSPs* [Larrosa and Meseguer, 1996]. Max-CSPs are a special case of the more general Weighted CSPs (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*.

In this paper we focus for simplicity on *Max-CSP*s. *Max-CSP* is an optimization problem with a search tree of bounded depth. Like other such optimization problems, the common 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 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 larger lower bounds.

The present paper improves on previous results by adding *Conflict-directed Backjumping* (CBJ) to the *B&B* algorithms presented in [Larrosa and Schiex, 2004] and in [Larrosa and Schiex, 2003]. Conflict-directed Backjumping (*CBJ*) [Prosser, 1993] is a method which is known to improve standard *CSP* algorithms [Dechter, 2003; Ginsberg, 1993; Kondrak and van Beek, 1997; Chen and van Beek, 2001]. In order to perform standard *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 [Chen and van Beek, 2001; Dechter, 2003].

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 reasoning 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

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

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 of 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. The maintenance of the data structures does not require additional iterations of the algorithm.

The results presented in this paper show that as reported for standard $CSPs$ by [Chen and van Beek, 2001], the improvement in run-time is dependent on the degree of the consistency maintenance procedure. However, the factor of improvement in the number of assignments when using CBJ is consistent and large.

Max-CSPs are presented in Section 2. A description of the standard *B&B* algorithm along with the maintenance procedures $NC*$, $AC*$ and $FDAC$ is presented in Section 3. Section 4 presents the *CBJ* algorithm for *B&B* with $NC*$, $AC*$ and $FDAC$. An extensive experimental evaluation, of the contribution of conflict directed backjumping to $B\&B$ with $NC*$, $AC*$ and $FDAC$, is presented in Section 5. Our conclusions are in Section 6.

2 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 $\{D_1, D_2, ..., D_n\}$ respectively. 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 variables. The $cost$ of a partial solution in a *Max-CSP* is the number of constraints violated by it (conflicts). 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.

3 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 *B&B* algorithm is presented in Figure 1. The general structure of the algorithm is different than its recursive presentation 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 hold for each successful assignment the state of the algorithm before it was performed. After each backtrack the current

B&B
1. $current\_state \leftarrow initial\_state$;
2. $CPS \leftarrow empty\_assignment$;
3. $i \leftarrow 0$;
4. **while**$(i \geq 0)$
5. **if**$(i = n)$
6. $upper\_bound \leftarrow lower\_bound$;
7. $i \leftarrow i - 1$;
8. **else foreach** $(a \in D_i)$
9. $temp\_state \leftarrow current\_state$;
10. **update_state**$(temp\_state, i, a)$;
11. **if**(**local_consistent**$(temp\_state)$)
12. $states[i] \leftarrow current\_state$;
13. $current\_state \leftarrow temp\_state$;
14. $i \leftarrow i + 1$;
15. $i \leftarrow$ **find_culprit_variable**();
16. $current\_state \leftarrow states[i]$;

update_state$(temp\_state, i, val)$
17. add (i, val) to $temp\_state.CPS$;
18. $temp\_state.CPS.cost \leftarrow temp\_state.CPS.cost + cost(i, val)$;
19. **for** $j \leftarrow (i + 1)$ to $(n - 1)$
20. **foreach** $(a \in temp\_state.D_j)$
21. **if**(conflicts($\langle i, val \rangle, \langle j, a \rangle$))
22. $cost(a, i) \leftarrow cost(a, i) + 1$;

Figure 1: Standard B&B algorithm

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. For simplicity of presentation we use a fixed order of variables and the function **find_culprit_variable** simply returns $i - 1$.

The naive and exhaustive $B\&B$ algorithm can be improved by increasing the level of the maintained local consistency. As a result, the value of the $lower\_bound$ of a *current partial solution* is increased. 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.

3.1 Node Consistency and NC*

Node Consistency is a very standard consistency maintenance method (analogous to Forward-checking 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 (CPS). For each value in a domain of an unassigned variable, one must determine whether 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 CPS. 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 CPS 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 as 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

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. $C_\phi \leftarrow C_\phi + c_j$;
6. $lower\_bound \leftarrow CPS.cost + C_\phi$;
7. **for** $j \leftarrow (i+1)$ to $(n-1)$
8. **foreach** $(a \in D_j)$
9. **if**$(a.cost + lower\_bound \geq upper\_bound)$
10. $D_j \leftarrow D_j \setminus a$;
11. **return** $(lower\_bound < upper\_bound)$;

Figure 2: Standard NC*

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 CPS's cost and C_ϕ.

Any value whose $lower\_bound$, i.e. the sum of the CPS's cost, C_ϕ and its own cost, exceeds the 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 2.

3.2 Arc Consistency and AC*

A stronger consistency maintenance procedure which is known to be effective for $CSPs$ is *Arc Consistency* (AC) [Bessiere and Regin, 1995].

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]. $AC*$ 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 domain has a value with a zero cost) [Larrosa and Schiex, 2004].

3.3 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]

[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].

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

4 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; Chen and van Beek, 2001]. 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 $global\ conflict\_set\ (GCS)$ is the set of assignments whose replacement can decrease the $lower\_bound$.*

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 $v_{i_j} \in D_i$, is the ordered list of assignments of variables in the $current\_partial\_solution$, which were assigned before variable i, and which conflict with v_{i_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 values of X_i (if the $conflict\_list$ of a value is shorter than c_i then all its assignments are included in the variables' $conflict\_set$).*

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-cost 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,

$update\_state(i, val)$
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. $GCS \leftarrow GCS \cup first\_element$ 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. $a.conflict\_list \leftarrow a.conflict\_list \cup (i, val)$;

find_culprit_variable(i)
12. **if**$(GCS = \phi)$
13. **return** -1;
14. $culprit \leftarrow$ latest assignment in GCS;
15. $GCS \leftarrow GCS \setminus culprit$;
16. **return** culprit;

Figure 3: The changes in $B\&B$ required for backjumping

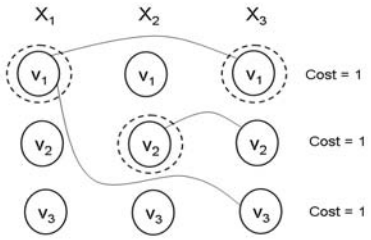

Figure 4: A conflict set of an $assigned$ variable

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 3 presents the changes needed for adding CBJ to standard *B&B*. After a successful assignment is added to the CPS, its cost is added to the cost of the CPS (lines 1,2). Then, 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 for function **find_culprit_variable** to do is to return the latest assignment (the one with the highest variable index) in the GCS (lines 14-16). In case the GCS is empty, function **find_culprit_variable** returns -1 and as a result, terminates the algorithm (lines 12,13).

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 bounds the space complexity of the algorithm's state by $O(n^2d)$. Since we hold up to n states, the overall space used is bounded by $O(n^3d)$.

Figure 4 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.

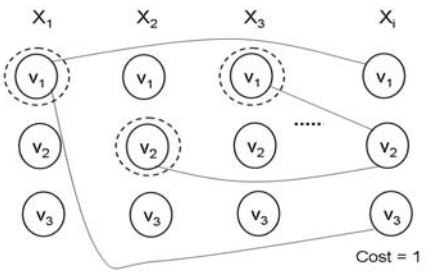

Figure 5: A conflict set of an $unassigned$ variable

However, replacing the current assignment of X_2 can lower the cost of value v_2 of variable X_3.

4.1 Node Consistency with CBJ

In order to perform CBJ 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 CPS which conflict with the corresponding value. The length of the $conflict\_list$ is equal to the cost of the value. Whenever $NC*$ 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 5 presents the state of an unassigned variable X_i. The CPS includes the assignments of three variables as in the example in Figure 4. 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 6 presents the changes in $NC*$ that are required 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). Note that the revised procedure uses the $conflict\_lists$ of removed values as well as of remaining values. For each domain D_i an additional set, $\hat{D}_i$ is maintained, which holds the values that were removed from D_i.

4.2 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 values' $conflict\_lists$ are added to the *global* $conflict\_set$. However, in $AC*$, costs

NC*_BJ(i)
1. **for** $j \leftarrow (i+1)$ to $(n-1)$
2. $c_j \leftarrow min\_cost(D_j)$;
3. **foreach** $(a \in (D_j \cup \hat{D}_j))$
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. **for** $j \leftarrow (i+1)$ to $(n-1)$
11. **foreach** $(a \in D_j)$
12. **if**$(a.cost + lower\_bound \geq upper\_bound)$
13. $\hat{D}_j \leftarrow \hat{D}_j \cup a$;
14. $D_j \leftarrow D_j \setminus a$;
15. **return** $(lower\_bound < upper\_bound)$;

Figure 6: Changes in $NC*$ that enable CBJ

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 CPS, 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 a removal of the last assignment in the CPS 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 CPS 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 variable's $conflict\_set$ 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 CPS is added to its $conflict\_list$. Caution must be taken when performing the assignment since the constant change in the 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 $priority\_cost$.

5 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 [Larrosa and Schiex, 2003; 2004].

Experiments were conducted on random *CSPs* of n variables, k values in each domain, a constraint density of p_1

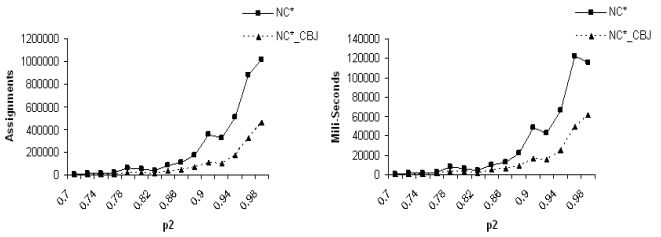

Figure 7: Assignments and Run-time of *NC** and *NC*_CBJ* (p_1 = 0.4)

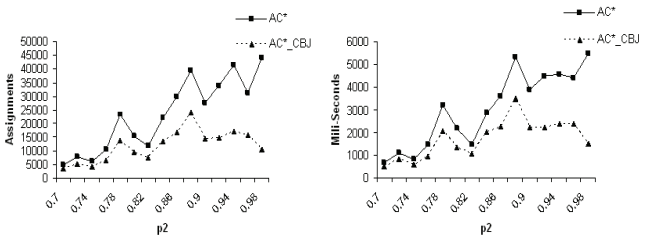

Figure 8: Assignments and Run-time of *AC** and *AC*_CBJ* (p_1 = 0.4)

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 CBJ to *B&B* algorithms using consistency maintenance procedures, the *B&B* algorithm with $NC*$, $AC*$ and $FDAC$ was 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 $p_1 = 0.9$.

The left hand side (LHS) of Figure 7 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 $p_2 = 0.99$. The right hand side (RHS) of Figure 7 shows similar results for cpu-time.

The LHS of Figure 8 presents the number of assignments performed 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. The RHS of Figure 8 presents the result in cpu-time which are similar but the difference is smaller than for the number of assignments. Note that both performance measures *decrease* for p_2 values above 0.98 for *AC*_CBJ*. This demonstrates a phase transition [Larrosa and Meseguer, 1996; Larrosa and Schiex, 2004].

Figure 9 presents similar results for the *AC** algorithm

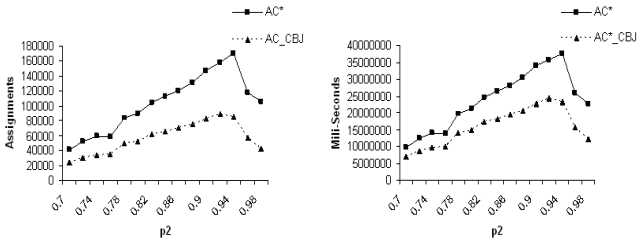

Figure 9: Assignments and Run-time of *AC** and *AC*_CBJ* ($p_1 = 0.9$)

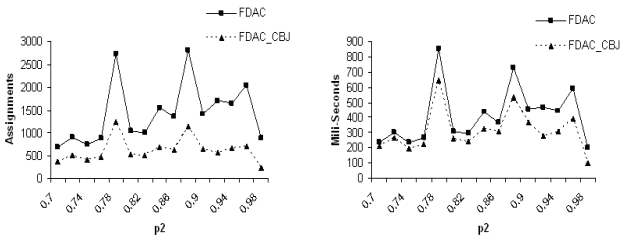

Figure 10: Assignments and Run-time of *FDAC* and *FDAC_CBJ* ($p_1 = 0.4$)

solving high density *Max-CSPs* ($p1 = 0.9$). The factor of improvement is similar to the low density experiments for both measures.

The LHS of Figure 10 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 on the RHS of Figure 10, is smaller than for the previous procedures. These differences are also presented for high density *Max-CSPs* in Figure 11.

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 algorithms 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.

6 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 the present study we adjusted conflict directed backjumping (*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 5 show that *CBJ* improves the performance of all versions of the $B\&B$ algorithm. The factor of improvement in the number of assignments when using conflict directed backjumping is consistent and large. The factor

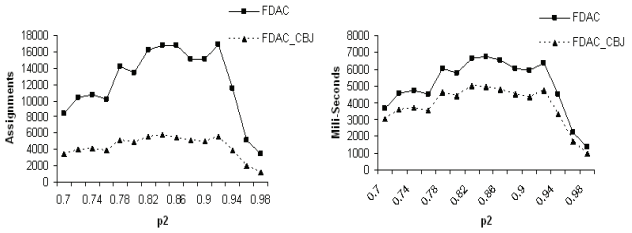

Figure 11: Assignments and Run-time of *FDAC* and *FDAC_CBJ* ($p_1 = 0.9$)

of improvement in run-time is dependent on the consistency maintenance procedure used (as reported for standard *CSPs* by [Chen and van Beek, 2001]). The factor of improvement does not decrease (and even grows) for random problems with higher density.

References

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

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

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

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

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

[Guéret *et al.*, 2000] 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.

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

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

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

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

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

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

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

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