Conflict based Backjumping for Constraints Optimization Problems

Roie Zivan and Amnon Meisels[⋆]
{zivanr,am}@cs.bgu.ac.il

Department of Computer Science,
Ben-Gurion University of the Negev,
Beer-Sheva, 84-105, Israel

Abstract. Constraints Optimization problems are commonly solved using a Branch and Bound algorithm enhanced by a consistency maintenance procedures [WF93] [LM96,LMS99,LS04]. 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-based Backjumping (CBJ) in Branch and Bound algorithms. The proposed algorithm maintains *Conflict Sets* which include only assignments whose replacement can lead to a better solution and backtracks according to these sets. CBJ can be added to Branch and Bound which uses the most advanced consistency maintenance heuristics, $NC*$ and $AC*$. The experimental evaluation of of $B\&B\_CBJ$ on random *Max-CSPs* shows that the performance of the algorithms are improved by a large factor.

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 are the scope of Max-Constraint Satisfaction Problems (*Max-CSPs*) [LM96]. Max-CSPs are a special case of the more general Weighted Constraints Satisfaction Problem (WCSPs) [LS04] in which each constraint is assigned with a weight which defines its cost if it is included in a solution. The weight of a solution is the sum of the weights of all conflicts (i.e. broken constraints) included in 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 *Constraints Optimization Problems*.

In this paper we focus for simplicity on *Max-CSP* problems. Since *Max-CSP* is an optimization problem with a limited search tree, the immediate choice for solving it is to use a *Branch and Bound* algorithm [Dec03]. In the last decade, various algorithms were developed for Max and Weighted CSPs [WF93,LM96,LMS99,LS04]. All of these algorithms are based on standard backtracking and gain their efficiency from the quality

[⋆] Supported by the Lynn and William Frankel center for Computer Sciences.

of the heuristic function (consistency maintenance procedure) they use. The best result for *Max-CSPs* was presented in [LMS99]. This result was achieved using a complex method which generates higher lower bounds by manipulating the order in which directional arc consistency is performed. In [LS04], the authors present new consistency maintenance procedures, *NC** and *AC** which improve on former versions of *Forward-checking* and *Arc-consistency*. However the performance of the resulting algorithms are close but do not outperform the Forward-checking method presented in [LMS99].

The present paper improves on previous results by adding *Conflict-based Backjumping* to the Branch and Bound algorithms presented in [LS04]. Conflict-based Backjumping (CBJ) is a method which is known to improve standard CSP algorithms [Dec03] [Gin93,ZM03]. 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.

Performing back-jumping for *Max-CSPs* is a much more complicated task 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 in to 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 results presented in this paper show that the above effort is worth while. Adding Conflict based Backjumping to Branch and Bound with $NC*$ and $AC*$ improves the runtime by a large factor.

Max-CSPs are presented in Section 2. A description of the standard *Branch and Bound* algorithm along with the $NC*$ and $AC*$ algorithm is presented in Section 3. The addition of CBJ to *Branch and Bound* with $NC*$ and $AC*$ is presented in Section 4. Section 7 introduces a correctness and completeness proof for *B&B_CBJ* with $NC*$ and $AC*$. An extensive experimental evaluation, which compares $B\&B$ with $NC*$ and $AC*$ to *B&B_CBJ* is presented in Section 8. The experiments were conducted on randomly generated *Max-CSPs*.

2 Distributed Constraint Satisfaction

A *Max - Constraint Satisfaction Problem* (*Max-CSP*) is composed, like a 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 $R \subseteq D_{i_k} \times D_{j_l} \times ... \times D_{m_n}$. 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 $< var, val >$, 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 an set of variables. The $cost$ of a partial solution in a *Max-CSP* is the number of conflicts included in it. An optimal **solution** to a *Max-CSP* is a partial solution that includes all variables and which includes a minimum 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 using 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 $upper\_bound$ is initialized to infinity and the $lower\_bound$ to zero. In each step of the algorithm, a partial solution, $current\_solution$, is expanded by assigning a value to a variable which is not included in it. After adding the new assignment, the $lower\_bound$ is updated with the cost of the updated $current\_solution$. The $current\_solution$ is expanded as long as the $lower\_bound$ is smaller than the $upper\_bound$. If a full solution is obtained, i.e. the $current\_solution$ includes assignments to all variables, the $upper\_bound$ is updated with the cost of the solution. If the $lower\_bound$ is equal 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.

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

3.1 Node Consistency and NC*

Node Consistency (or Forward-checking) is a very standard consistency maintenance method in standard $CSPs$ [Tsa93,Dec03]. 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 $current\_solution$. In *Max-CSPs*, for each value in a domain of an unassigned variable, one must determine if assigning it to its variable 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 $current\_solution$. 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 $current\_solution$ 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 [LS04], 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

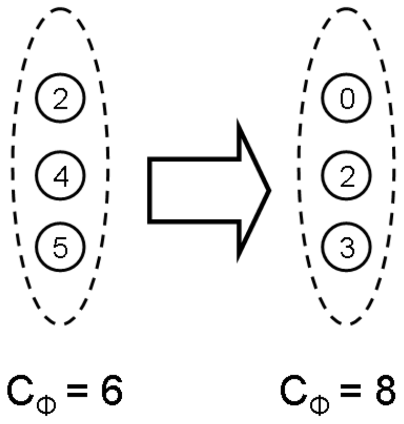

Fig. 1. Values of a variable before and after running NC*

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 $current\_solution$'s cost and C_ϕ.

Figure 1 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 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 was raised by 2.

Any value whose $lower\_bound$, i.e. the sum of the $current\_solution's$ 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 [LS04].

3.2 Arc Consistency and AC*

Another consistency maintenance procedure which is known to be effective for $CSPs$ is *Arc Consistency*. In standard $CSPs$, Arc-Consistency is more restricted than NC, for eliminating inconsistent values from future variables. The idea of standard AC [BR95] is that if a value v of some unassigned variable X_i, is in conflict with all values 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* Arc-Consistency is used to project costs of conflicts between unassigned variables, over values costs. As for standard $CSPs$, a value in a domain of

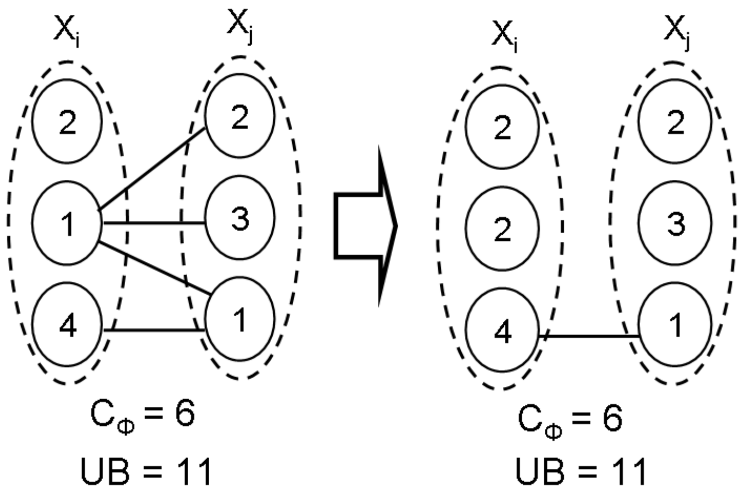

Fig. 2. Values of a variable before and after running NC*

an unassigned variable, which is in conflict with all the values of another unassigned variable, will cause a conflict when it is assigned. This information is used in order to increment the cost of the value. Values for which the sum of their cost and the global $lower\_bound$ exceeds the $upper\_bound$, are removed from their variable's domain. However, in AC every removal of a value can cause an increase in the cost of another value. Therefore, an additional check has to be made.

$AC*$ 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 for AC.

Figures 2 and 3 present an example of the AC* procedure. On the LHS of Figure 2 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 2. The left hand side of Figure 3 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. After the incrementation of C_ϕ, the 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 3.

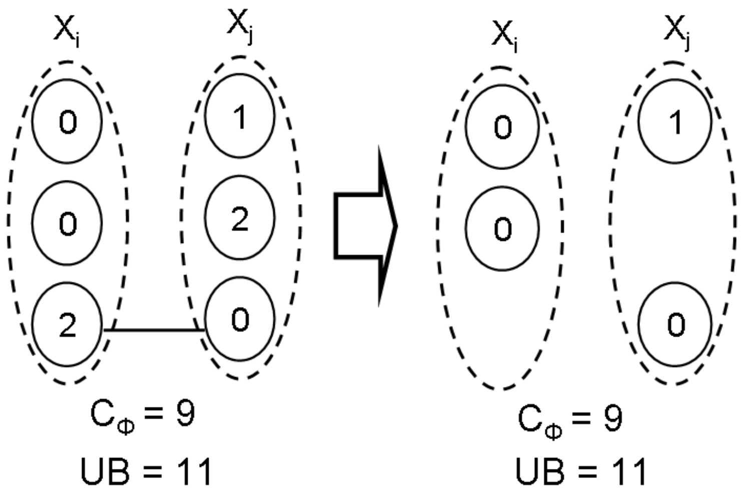

Fig. 3. Values of a variable before and after running NC*

4 Branch and Bound with CBJ

The addition of Backjumping to standard CSP search is known to improve the run-time performance of the search by a large factor [Dec03,Gin93]. The various algorithms which perform backjumping differ by the method of resolution which is used to determine the selected variable for the algorithm to backjump to. The common choice is to maintain 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 selected to backtrack to 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 [Pro93].

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, the construction and maintenance of these conflict sets are a much more complicated task. In the simplest version of $B\&B$, the $lower\_bound$ of a $current\_solution$ is its $current\_cost$ (i.e. the number of conflicts it contains). The algorithm backtracks only when this cost is larger or equal to the $upper\_bound$. When a backtrack operation is performed, the goal is to decrease the cost by replacing an assignment. More specifically, every binary constraint is between an $earlier$ variable, which is the variable that was assigned first and the second variable of the constraint which was assigned $later$. If we assume that for every variable the first value to be assigned is the one with minimal number of conflicts (i.e. the value with a minimal cost), then backtracking to a $later$

variable cannot improve the cost of the $current\_solution$ (see the proof in section 7). The only way that the cost of a $current\_solution$ can be lowered is by backtracking to an $earlier$ variable and replacing its assignment. In order to keep the completeness of the algorithm, the backtrack operation must be performed to the last assigned variable in the group of candidate $earlier$ assignments

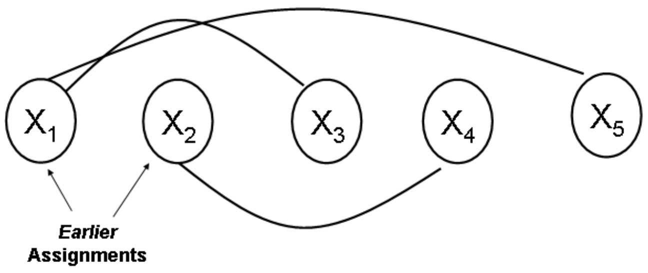

Fig. 4. Values of a variable before and after running NC*

Figure 4 presents a partial solution with 5 variables and cost 3. The conflict set of this partial solution includes the assignments of X_1 that is in conflict with the assignment of X_3 and X_5, and X_2 which is in conflict with the assignment of X_4. Although there are three conflicts in this example, only variables X_1 and X_2 are $earlier$ in all three conflicts. Therefore the conflict set of this partial solution must include both of them. In standard $CSPs$, when backtracking from variable X_5, it would be enough to check its *conflict set* in order to choose the variable to backtrack to. This example shows why generating the conflict set only according to the last variable state as done in standard $CSPs$ in not sufficient in the case of *Max-CSPs*. Variable X_5 is in conflict only with the assignment of X_1 however, the cost of the partial solution can be lowered by backtracking to X_2.

Unfortunately, generating the conflict set out of all $earlier$ assignments of the conflicts in the $current\_solution$ is not enough. In every $later$ assigned variable in each conflict, unassigned values may have conflicts with different $earlier$ assignments. For example, if the value assigned to a variable X_i, $i > 2$, in the $current\_solution$ has a cost 1, and the $earlier$ assignment with whom it has a conflict is the assignment of X_1, X_1 is added to the conflict set. However if there is an unassigned value in the domain of X_i also with cost 1 but whose conflict is with the assignment of variable X_{i-1}, the algorithm must backtrack to X_{i-1} which was assigned later than X_1. In order to give a formal description of the construction of the conflict set, the following definitions are needed:

Definition 1 *A $conflict\_list$ of value v_j from the domain of variable X_i, is the list of assignments in the $current\_solution$ of variables which were assigned before i, and v_j has conflicts with. The assignments in the conflict list are in the same order the assignments in the $current\_solution$ were performed.*

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 *The $conflict\_set$ of variable X_i with cost c_i is the union of the first c_i assignments in the $conflict\_list$ of all its values.*

Definition 4 *A* global $conflict\_set$ *is the set of assignments such that the algorithm back-jumps to the latest assignment of the set.*

In the case of simple $B\&B$, the *global* $conflict\_set$ is the union of all the $conflict\_sets$ of all assigned variables. Another way to explain this need of adding the conflicts of all values and not just the conflicts of the assigned value, is that in order to decrease the cost of the $current\_solution$, a value which has less conflicts should be able to be assigned. Therefore, the latest assignment that can be replaced, and possibly decrease the cost of one of the variables values to be smaller than the variables current cost should be considered.

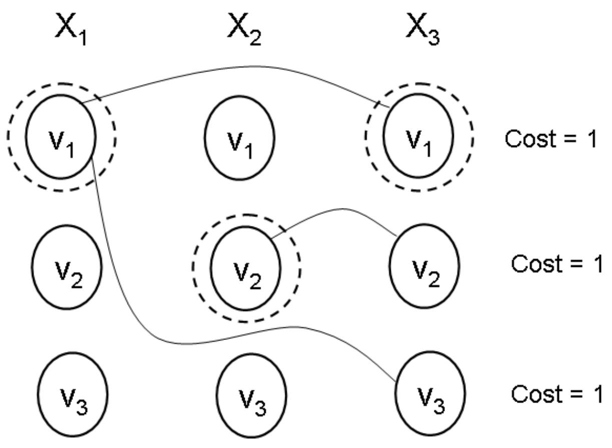

Fig. 5. A conflict set of an $assigned$ variable

Figure 5 presents the state of three variables which are included in the $current\_solution$. 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 conflicted with the assignment of X_2 since replacing it can lower the cost of value v_2 of variable X_3.

5 Node Consistency with CBJ

In order to perform conflict based backjumping in a $B\&B$ algorithm using 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 $current\_solution$ 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 minimum 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 since backtracking to the head of their list can cause their return 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.

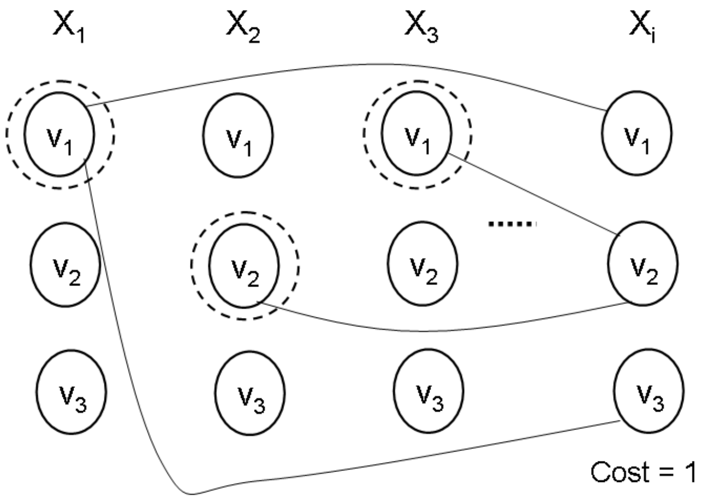

Fig. 6. A conflict set of an $unassigned$ variable

Figure 6 presents the state of an unassigned variable X_i. The $current\_solution$ includes the assignments of three variables as in the previous example. 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*.

6 Arc Consistency 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 and the correlation between the value's cost and the number of conflicts it has with the assignments in the $current\_solution$ (i.e. the length of its $conflic\_list$) does not hold. In order to find the right conflict set in this case one must keep in mind that except for an empty $current\_solution$,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 (see Section 7). This means that replacing the last assignment in the $current\_solution$ would return the value which is not in conflict with v_k, to the domain of X_j. This is enough to decrease the cost of the value v_k. Whenever a cost of a value is raised by arc consistency, the last assignment in the $current\_solution$ must be added to the end of the value's $conflict\_list$. By maintaining this property in the $conflict\_list$ the variables $conflict\_set$ and the *global conflict_set* can be generated in the same way as for $NC*$.

7 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. Consider the case that a $current\_solution$ has a length k and the index of the variable of the latest assignment in the $current\_solution$'s corresponding $conflict\_set$ is l. Assume in negation, that there exists an assignment in the $current\_solution$ with a variable index $j > l$, that by replacing it the cost of a $current\_solution$ 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 all their value's $conflict\_lists$. 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 means that the variables costs stay the same and the cost of the $current\_solution$ too in contradiction to the assumption. $\square$

Next, we prove the consistency of the *global conflict_set* in $B\&B\_CBJ$ with the $NC*$ consistency maintenance procedure. The above proof holds for the assignments added due to conflicts within the $current\_solution$. For assignments added to the *global conflict_set* due to conflicts of unassigned variables with assignments in the $current\_solution$ 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 is zero. If some assignment of variable X_j in the $current\_solution$ was not added to the *global*

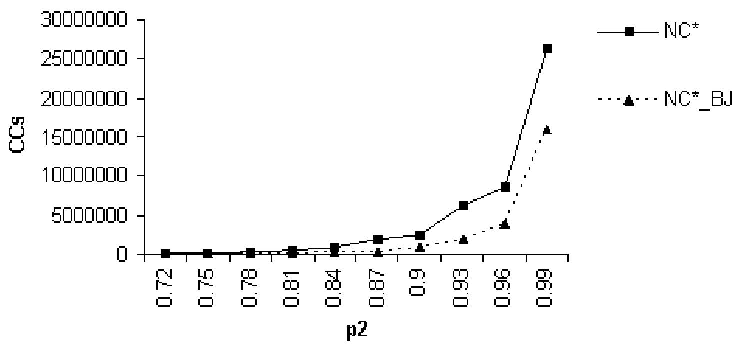

Fig. 7. Number of constraints checks performed by $NC*$ and $NC*\_BJ$ on low density Max-CSPs (p_1 = 0.4)

conflict_set it means that it was not a prefix of any $conflict\_list$ of 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 $lower\_bound$. □

Having established the correctness of the $conflict\_set$ for the $current\_solution$ of a Branch and Bound algorithm and for the $NC*$ procedure, the consistency of the *global conflict_set* for $AC*$ is immediate. The only difference between $NC*$ and $AC*$ is the addition of the last assignment in the $current\_solution$ 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 which is left out of the paper, proves that at any step of the algorithm, only a removal of a value can cause an increment of some value's cost due to arc consistency. □

8 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 the total number of constraints checks [Dec03].

Experiments were conducted on random constraints satisfaction problems of n variables, k values in each domain, a constraints density of p_1 and tightness p_2 (which are commonly used in experimental evaluations of CSP algorithms [Smi96]). In all of the experiments the *Max-CSPs* included 10 variables ($n = 10$), 10 values for each variable ($k = 10$). Two values of constraints density $p_1 = 0.4$ and $p_1 = 0.7$ were used to generate the *Max-CSPs*. The tightness value p_2, was varied between 0.72 and 0.99, since the hardest instances of *Max-CSPs* are for high p_2 [LM96]. For each pair of fixed density

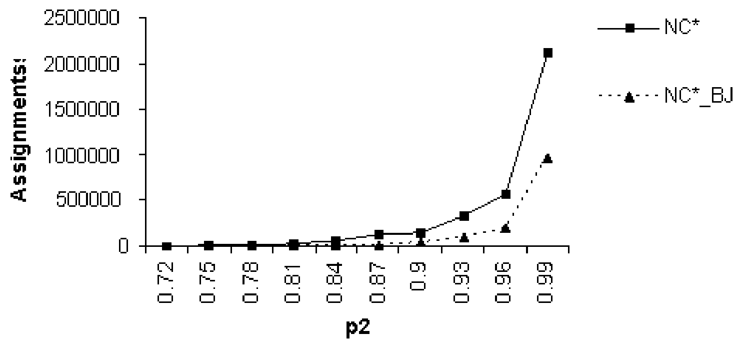

Fig. 8. Number of assignments performed by $AC*$ and *AC*_BJ* on low density Max-CSPs (p_1 = 0.4)

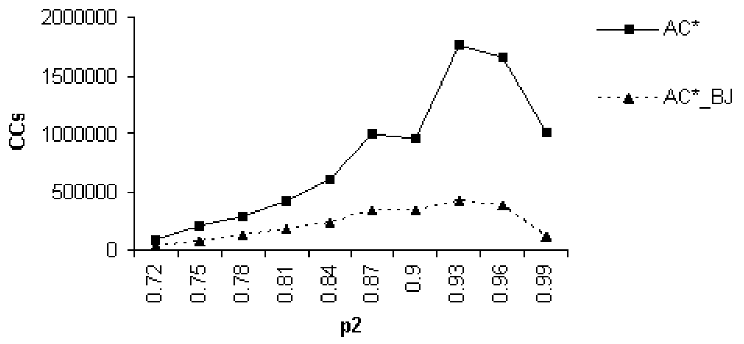

Fig. 9. Number of constraints checks performed by $AC*$ and *AC*_BJ* on low density Max-CSPs (p_1 = 0.4)

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 based Backjumping* to *Branch and Bound* algorithms using consistency maintenance procedures the $B\&B$ algorithm with $NC*$ and $AC*$ procedures were implemented. The results presented show the performance of these algorithms with and without CBJ.

Figure 7 presents the computational effort in number of constraints checks to find a solution, performed by $NC*$ and *NC*_BJ*. For the hardest instances, where p_2 is higher

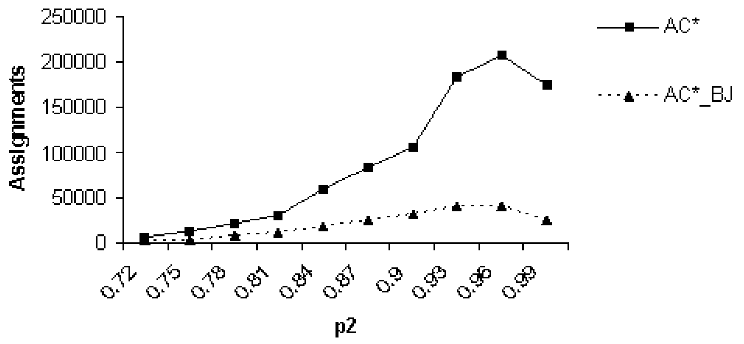

Fig. 10. Number of assignments performed by $AC*$ and *AC*_BJ* on low density Max-CSPs (p_1 = 0.4)

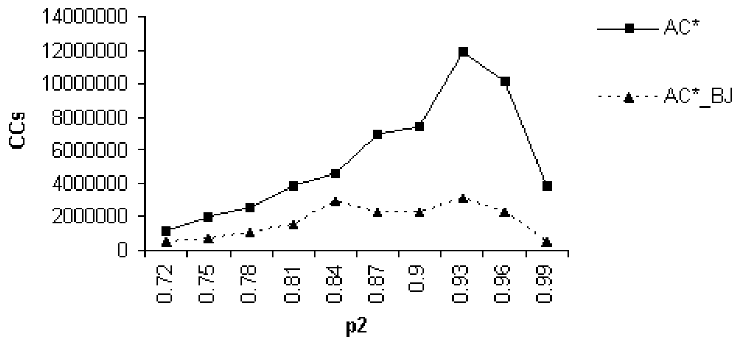

Fig. 11. Number of constraints checks performed by $AC*$ and *AC*_BJ* on high density Max-CSPs (p_1 = 0.7)

than 0.9, *AC*_BJ* outperforms $AC*$ by a factor of between 5 at $p_2 = 0.93$ and 2 at $p_2 = 0.99$. Figure 8 shows similar results in the number of assignments performed by the algorithms.

Figure 9 presents the computational effort in number of constraints checks to find a solution, performed by $AC*$ and *AC*_BJ*. For the hardest instances, where p_2 is higher than 0.9, *AC*_BJ* outperforms $AC*$ by a factor of 5. Figure 10 shows similar results in the number of assignments performed by the algorithms.

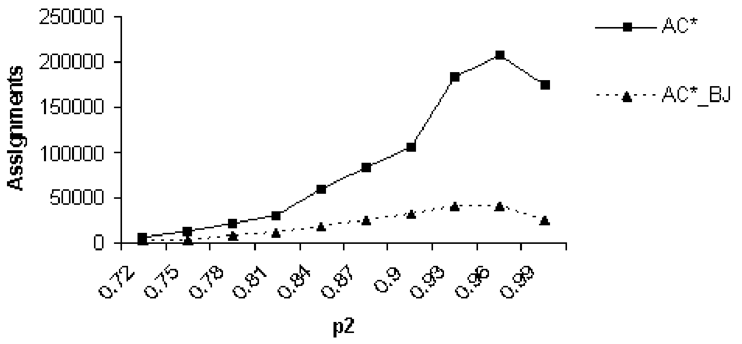

Fig. 12. Number of assignments performed by $AC*$ and *AC*_BJ* on high density Max-CSPs (p_1 = 0.7)

Figure 11 and 12 show similar results for the $AC*$ algorithms solving high density *Max-CSPs* ($p1 = 0.7$). Interestingly although the scale is much higher the factor remains the same.

9 Discussion

Conflict based Backjumping is a powerful technique used to improve the run-time of standard CSP algorithms [Pro93,Dec03,Gin93]. The experimental results, show that this is true for Branch and Bound algorithms with consistency maintenance procedures. These results might come as a surprise because unlike in standard $CSPs$, the conflict sets in *B&B_CBJ* are constructed by the union of conflicts of unassigned values as well as assigned values. This means that the number of values which their conflicts are taken into consideration is larger than when performing CBJ for standard $CSPs$. Noting this fact, one could expect the maintained *global conflict_set* to be larger and consequentially have a smaller effect. This assumption is proven wrong by the result presented in this paper.

A possible explanation is the properties of the hard instances of *Max-CSPs*. In contrast to standard $CSPs$, where the hardest instances are approximately in the center of the range of p_2 (about 0.5, depends on the exact value of p_1) [Smi96], the hardest instances of *Max-CSPs* are when p_2 is close to 1.0 [LM96]. For high values of p_2 when some assignment is added to the $conflict\_list$ of some value, it is very probable that it would also be added to the $conflict\_lists$ of the other values of the same variable. Therefore when we add the prefix of size c_i of all values in the domain of X_i to the variable's $conflict\_set$ in many cases these prefixes are very similar if not identical. This keeps the $conflict\_set$ small and generates non-trivial jumps.

10 Conclusions

Branch and Bound is the most common algorithm used for solving *Max-CSPs*. Former studies improved the results of the Branch and Bound algorithms by improving the consistency maintenance procedure used by the algorithm [WF93,LM96,LMS99,LS04]. In this study we adjusted *Conflict-based Backjumping* which is a common technique in standard CSP algorithms to Branch and Bound with extended consistency maintenance procedures. The results presented in Section 8 are striking. CBJ improves the performance of the *Max-CSP* algorithm by a large factor. The factor of improvement does not decrease for problems with higher density.

References

[BR95] C. Bessiere and J.C. Regin. Using bidirectionality to speed up arc-consistency processing. *Constraint Processing (LNCS 923)*, pages 157–169, 1995.

[Dec03] Rina Dechter. *Constraints Processing*. Morgan Kaufman, 2003.

[Gin93] M. L. Ginsberg. Dynamic backtracking. *J. of Artificial Intelligence Research*, 1:25–46, 1993.

[LM96] J. Larrosa and P. Meseguer. Phase transition in max-csp. In *Proc. ECAI-96*, Budapest, 1996.

[LMS99] J. Larrosa, P. Meseguer, and T. Schiex. Maintaining reversible dac for max-csp. *Artificial Intelligence*, 107:149–163, 1999.

[LS04] J. Larrosa and T. Schiex. Solving weighted csp by maintaining arc consistency. *Artificial Intelligence*, 159:1–26, 2004.

[Pro93] P. Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9:268–299, 1993.

[Smi96] B. M. Smith. Locating the phase transition in binary constraint satisfaction problems. *Artificial Intelligence*, 81:155 – 181, 1996.

[Tsa93] E. Tsang. *Foundations of Constraint Satisfaction*. Academic Press, 1993.

[WF93] R. J. Wallace and E. C. Freuder. Conjunctive width heuristics for maximal constraint satisfaction. In *Proc. AAAI-93*, pages 762–768, 1993.

[ZM03] R. Zivan and A. Meisels. Synchronous vs asynchronous search on discsps. In *Proc. 1st European Workshop on Multi Agent System, EUMAS*, Oxford, December 2003.