Partial ARTIAL Incompatible based Lower Bound of NC* For MAX-CSPs

Ashraf M. Bhery, Soheir M. Khamis, and Wafaa A. Kabela

Division of Computer Science, Department of Mathematics,
Faculty of Science, Ain Shams University, Cairo, Egypt.
bhery_as@yahoo.com, soheir_khamis@hotmail.com, wafaa_ai2000@yahoo.com

Abstract

Maximal Constraint Satisfaction Problems (Max-CSPs) are constraint optimization problems, in which the goal is to maximize the number of satisfied constraints. Max-CSPs are, in general, solved using Branch and Bound (B&B) algorithms. Their respective efficiency highly depends on the quality of the lower bound. The naive B&B has been improved by using consistency maintenance procedures and conflict backjumping. In this paper, the authors give a new treatment for improving NC*-CBJ algorithm for solving a Max-CSP which is the B&B algorithm using advanced Node Consistency procedure (NC*) and performing Conflict-directed Backjumbing (CBJ), [17]. The goal of this improvement is increasing the lower bound of NC*-CBJ via taking into account more inconsistencies which resulted from the proposed partially incompatible relation between the future variables. The introduced treatment leads to suggesting new algorithm, M-NC*-CBJ, which is a natural successor of NC*-CBJ algorithm including the modification of the lower bound. By comparing with the results of NC*-CBJ, the experimental results of M-NC*-CBJ on random CSPs show improvement both in execution times and number of assignments.

Keywords: *Constraint satisfaction; Max-CSP; Branch and Bound; Lower bound; Node consistency; NC*-CBJ.*

1. Introduction

The Constraint Satisfaction Problem (CSP) is a powerful and efficient framework for modeling and solving many real world problems. Some well known examples are: Scheduling, planning, network management and configuration. Due to the ability of (CSP) techniques it has helped in the development of tackling some recent applications including: Computer graphics (expressing geometric coherence in the case of scene analysis, drawing programs, user interfaces), natural language processing (construction of efficient parsers), database systems (to ensure and/or restore consistency of the data), molecular biology (DNA sequencing, chemical hypothesis reasoning), business applications (option trading), electrical engineering (to locate faults), circuit design (to compute layouts) and transport problems.

The Constraint Satisfaction Problem (CSP) framework allows researchers to define problems in a declarative way, quite independently from the solving process. However, when the problem is over-constrained, the answer no solution is generally unsatisfactory. In this case, the goal is, often, to find an assignment satisfying as many constraints as possible which is known as a Maximal constraint satisfaction problem (Max-CSP). A Max-CSP is a constraint problem whose solutions maximize the number of satisfied constraints [2].

Many algorithms have been developed for Max-CSP using Branch and Bound (B&B) methods combined with heuristics for selecting the variables and the values at each step of the search tree, e.g., [2, 4, 14]. Their respective efficiency highly depends on the quality of the lower bound. Many lower bound computation methods for branch and bound have been developed for solving Max-CSP see e.g., [2, 3, 5, 15]. Those methods can be considered as procedures that search for disjoint inconsistent constraints in the Max-CSP instance under consideration. The difference among them is the technique used to detect inconsistencies. In [2], the authors computed a lower bound of inconsistencies from the set of assigned variables. Furthermore, the effect of these variables on unassigned ones by using forward checking. This lower bound is improved by including inconsistencies among future variables by the usage of Directed Arc consistency (DAC) [15]. Further improvements have been introduced in [3] and [16]. Recent works focus on the exploitation of propagation mechanisms to improve the value of the lower bound, through soft arc-consistency algorithms ([6, 11, 12], initially introduced in [1]) or by using conflict-set based algorithms [10]. These algorithms detect violations which are ignored by the previous reference algorithm PFC-MRDAC [5].

One of the most successful approaches to build lower bounds has been obtained by extending some local consistency notions to weighted CSP [1]. In [7], Larrosa and schiex presented new consistency maintenance procedures based on their work which introduced in [6]. Some examples of this procedures are NC* for Node Consistency and AC* for Arc Consistency. In [17], NC*-CBJ algorithm was obtained by adding conflict directed backjumping (CBJ) to branch and bound, which maintains NC*. This addition leads to improving the performance of branch and bound algorithm for solving Max-CSPs.

In this paper, the proposed modification is to improve a lower bound of NC*-CBJ. Its idea depends on taking into account more inconsistencies which resulted from the given partially incompatible relation between the future variables. NC*-CBJ algorithm that including the modification of the lower bound, leads to suggesting M-NC*-CBJ algorithm. By comparing with the results of NC*-CBJ, the experimental results of M-NC*-CBJ on random CSPs show improvement both in CPU-time s and number of assignments.

This paper is organized as follows. In section 2, some preliminaries and definitions required in the rest of the paper are introduced. Section 3 includes a brief description of iterative NC*-CBJ algorithm which is B&B algorithm combined with NC* and CBJ. In section 4, the new partially incompatible relation is defined. Furthermore, the modified NC*-CBJ algorithm that depends on the new lower bound, M-NC*-CBJ, is described. Section 5 includes analysis of complexity of the additional part to NC*-CBJ lower bound and its correctness. Moreover, we prove that M-NC*-CBJ lower bound is still lower bound. Section 6 contains experimental results of M-NC*-CBJ showing a clear performance improvement in both execution time and number of assignments. Finally, section 7 presents the conclusion of the paper and some directions of future work.

[1] Max-CSP is a special case of weighted CSP in which each constraint has an associated weight, and the goal is to find a solution that maximizes the total weight of the satisfied constraints.

2. Basic Definitions

This section introduces the main notations and definitions that are used throughout the paper. In the following, the definitions of a Maximal Constraint Satisfaction problem (Max-CSP) and the local Node Consistency property (NC*) used for solving a Max-CSP are given.

A classical binary Constraint Satisfaction Problem (CSP) is a triple $Z = (X, D, R)$, where $X = \{1, 2, ..., n\}$ is a set of variables, $D = \{ D_1, D_2, ..., D_n \}$ is a set of domains of the variables in X, and R is a set of (possibly) unary or binary constraints defined on subsets (one or two) of variables of X. Each variable $i \in X$ has a finite domain $D_i \subseteq D$ of values that can be assigned to it. (i, a) denotes the assignment of value $a \in D_i$ to variable i. A partial solution, t, is an ordered set of values assigned to the ordered set of variables $X_t \subseteq X$ (namely, the k-th element of t is the value assigned to the k-th element of X_t). A unary constraint R_i is a subset of D_i containing the permitted assignments to variable i. A binary constraint R_{ij} is a set of pairs from $D_i \times D_j$ containing the permitted simultaneous assignments to i and j. Binary constraints are symmetric, i.e., $R_{ij} \equiv R_{ji}$. The set of (one or two) variables whose values are restricted by a constraint is called its scope. A partial solution t is *consistent* if it satisfies all constraints whose scope are included in X_t. It is *globally consistent* if it can be extended to a complete consistent assignment. A *solution* to CSP consists of finding a consistent complete assignment [13]. In some cases, a CSP instance may be over-constrained and thus, admits no such solution. We can then be interested in finding a complete instantiation that best respects the set of constraints. In this work, we consider the Maximal Constraint satisfaction Problem. An optimal solution of a Max-CSP is a complete instantiation satisfying maximum number of constraints, i.e., violating the minimum number of constraints [2].

Existing algorithms for solving Max-CSP are basically designed to follow a branch and bound ($B\&B$) schema. These algorithms perform successive assignments of values to variables through a depth-first traversal of the search tree. In the search tree, internal nodes represent partial assignments of values to variables. In addition, a leaf that ends a branch of $|X|$ nodes stands for a complete assignment. At each node, assigned variables are called Past $(P)^2$ while unassigned variables are called Future (F). B&B algorithms associate a cost to each node in the search tree. The cost of a node is the number of constraints violated by its assignment. In the following, the formal definition of the cost of the current partial solution (CPS) is given.

2 If t is partial assignment, then $X_t = P$

Definition 2.1. *Cost(CPS) =* $\sum_{i,j \in P, i<j} r_{ij}(a,b)$ *such that*

$$r_{ij}(a,b) = \begin{cases} 1 & if\ \exists R_{ij} \wedge (a,b) \notin R_{ij}, \\ 0 & otherwise, \end{cases}$$

where a and b are the values assigned to past variables i and j, respectively.

The main idea of B&B is simple and clear. In a B&B algorithm, two bounds are constantly maintained which are determined during execution. An Upper Bound (UB) is the cost of the best solution found so far and is initialized to infinity. A Lower Bound (LB) is an underestimation of the minimal number of constraints that will become unsatisfied if the current partial assignment is completed. At every node, the B&B algorithm compares the UB with the LB. If LB $\geq$ UB, the algorithm prunes the subtree below the current node and then backtracks (backjumping) to a higher level in the search tree. If LB $<$ UB, the algorithm tries to find a better solution by extending the current partial solution by instantiating one more variable. A current partial-solution, CPS, is expanded by assigning a value to a variable[3] which is not included in it.

In the description of the maintenance local consistency and CBJ in a B&B-NC*-CBJ algorithm, the following definition is used. In the definition, $\oplus$ represents a list concatenation

Definition 2.2. *For i* $\in$ *F and a* $\in D_i$ *, the cost of (i, a),* $C_i(a)$*, is the number of past variables which its assignments are inconsistent with (i, a) which is determined by*

$$C_i(a) = \sum_{j \in p} r_{ij}(a,b),$$

where b is the value assigned to past variable j.

Additionally, the conflict-list of (i, a), $L_i(a)$, is the ordered list which includes all the assignments in the current solution that inconsistent with (i, a) which is given by

$$L_i(a) = \oplus_{j \in P, r_{ij}(a,b)=1} (j,b),$$

and the length of $L_i(a)$ is equal to $C_i(a)$.

By using the previous definition, the current-cost and the conflist-set of a variable[4] can now be defined. In case of an assigned variable, its current cost is the cost of its assigned value; otherwise, the current-cost is the minimal cost of a value in its current domain.

[3] Values are always assigned using the M-cost heuristic i.e. the next value to be assigned is the value with the smallest cost in the variable's current domain.

[4] Definition 2.3 and 2.4 are informally founded in [17].

Definition 2.3. *Given a variable i the current-cost of i, C-cost(i), is given by*

$$C-cost(i) = \begin{cases} C_i(a) & if \quad (i \in P) \wedge (a \text{ is the assigned value to } i), \\ min_{a \in D_i} C_i(a) & if \ i \in F. \end{cases}$$

Definition 2.4. *Let an integer value c be the cost of variable i. A conflict-set, S-conflict(i, c), of i with cost c is given by*

$$\text{S-conflict}(i, c) = \bigcup_{a \in D_i} first(c, L_i(a)).$$

Where the function first(c, $L_i(a)$) returns the first (most recent) c assignments in $L_i(a)$. If length of $L_i(a)$ is less than c, then the function returns all assignments in $L_i(a)$.

Max-CSPs are usually solved with a tree produced from branch-and-bound in which each node is a partial solution. To accelerate the search, local consistency properties are widely used to transform the sub-problem at each node of the tree to an equivalent simpler one. The simplest local consistency property is the following advanced node consistency (NC*)[5]. For defining NC* property, assume the existence of a zero-arity constraint, $C_\varnothing$, whose initial value is equal to zero.

Definition 2.5. *Let Z = (X, D, R) be a binary Max-CSP and k > $|R|$; the number of constraints in R. A variable i is node-consistent iff*

i) $\forall \ a \in D_i, (C_\varnothing + C_i(a)) < k.$
ii) $\exists \ a \in D_i$ such that $C_i(a) = 0.$

Such a value a is support for the variable node consistency. Then, A Max-CSP is node consistence iff every variable is node consistent.

Obviously, the property of node consistency can be enforced in time and space O(*nd*), where *n* is number of the variables and *d* is the maximum domain size.

3. Solving Max-CSP by B&B-NC*-CBJ

In this section, we consider an improved version of the B&B algorithm which is known by NC*-CBJ [17], for Max-CSP. NC*-CBJ maintains local consistency procedure (NC*) with conflict-directed backjumbing (CBJ). In the following, the pseudo-code of B&B-CBJ algorithm is given. This algorithm is iterative and it includes functions that perform consistency checking and conflict-directed backjumbing.

[5] This definition is a special case of the general definition introduced in [6] for Weighted CSP. The NC* considered in this paper is for Max-CSP.

Algorithm (1)

```
Procedure B&B-CBJ
1:  begin
2:  current state ← initial state;
3:  CPS ← empty-assignment;
4:  GCS ← empty-assignment;
5:  $i \leftarrow 0$;
6:  while $i \geq 0$ do
7:     if $i = n$ then
8:        Upper-Bound ← Lower-Bound ;
9:        $i \leftarrow i - 1$;
10:    else
11:       for all $a \in D_i$ do
12:          temp-state ← current-state;
13:          update-state ($i$, $a$);
14:          if local-consistent($i$) then
15:             states[$i$] ← current-state;
16:             current-state ← temp-state;
17:             $i \leftarrow i + 1$;
18:          end if
19:       end for
20:    end if
21:    $i$ ← find-culprit-variable( );
22:    current-state← states[$i$ ];
23: end while
24: end.
```

In the B&B-CBJ algorithm, a Global Conflict Set (GCS) that includes the union of the conflict sets of all assigned and unassigned variables is also maintained. The B&B-CBJ algorithm consists of procedure update-state, function local-consistency, and function find-culprit-variable. The description of function find-culprit-variable is given in Algorithm(2) that is activated when the algorithm B&B-CBJ executes a backjump step. Find-culprit-variable returns the latest assignment (the one with the highest variable index) in the GCS. When the GCS is empty, function find-culprit-variable returns -1 and hence the execution of B&B-CBJ algorithm will terminate. Procedure update-state( ) and function local-consistent( ) are described as follows.

Algorithm (2)

<u>**Function find-culprit-variable(*i*)**</u>

```
1:  begin
2:   if GCS = $\emptyset$ then
3:     return -1 ;
4:   end if
5:   culprit ← latest assignment in GCS ;
6:   GCS ← GCS \ culprit ;
7:   return culprit ;
8:  end.
```

Update with CBJ

In this part, we present the procedure update-state( ) which take the new assignment as input. In Lines 2,3 from this procedure the new assignment is added to the Current Partial Solution (CPS) and the cost(CPS) is increased by the value C_i (val), respectively. Then, Lines (4-9) are used to add a conflict set of assigned variable *i* to GCS. Finally, Lines (10-17) are designed to update the cost and conflict-list of each value in the domain of future variables according to the added new assignment.

Algorithm (3)

<u>**Procedure update-state(*i, val*)**</u>

```
1:  begin
2:   add ($i$, val) to CPS;
3:   cost(CPS)← cost(CPS) + $C_i(val)$;
4:   for all $a \in D_i$ do
5:     for 1 to $C_i(val)$ do
6:        GCS ←GCS ∪ first-element in ($L_i(a)$);
7:        delete first-element from ($L_i(a)$);
8:     end for
9:   end for
10:  for $j = i + 1$ to $n - 1$ do
11:      for all $a \in D_j$ do
12:         if conflicts(($i$, $val$), ($j$, $a$ )) then
13:            $C_j(a) \leftarrow C_j + 1$;
14:            $L_j(a) \leftarrow L_j(a) \oplus (i, val)$;
15:         end if
16:      end for
17:  end for
18:  end.
```

NC*-CBJ

In general, a local-consistency function is used to reduce domains of unassigned variables. In the next, we restricted to advanced node consistency with CBJ named NC*-CBJ, as described in [17]. In NC*-CBJ, the algorithm computes a global lower bound of the number of unsatisfied constraints as the cost of current partial solution plus sum of minimum cost of each future variable. Algorithm(4) enforces NC* and maintains CBJ.

Algorithm (4)
<u>Function NC*- CBJ(*i*)</u>

```
1:  begin
2:  for $j \leftarrow i + 1$ to $n - 1$ do
3:      $c_j \leftarrow$ C-cost($j$);
4:      for all $a \in (D_j \cup \hat{D}_j)$ do
5:          $C_j(a) \leftarrow C_j(a) - c_j$ ;
6:          for 1 to $c_j$ do
7:              GCS $\leftarrow$ GCS $\cup$ first-element($L_j(a)$);
8:              delete first-element($L_j(a)$);
9:          end for
10:     end for
11: $C\emptyset \leftarrow C\emptyset + c_j$ ;
12: end for
13: lower bound $\leftarrow$ cost(CPS) + $C\emptyset$;
14: for $j \leftarrow i + 1$ to $n - 1$ do
15:     for all $a \in D_j$ do
16:         if $C_j(a)$ + Lower-Bound $\geq$ Upper-Bound then
17:             $\hat{D}_j \leftarrow \hat{D}_j \cup \{a\}$;
18:             $D_j \leftarrow D_j \setminus \{a\}$;
19:         end if
20:     end for
21: end for
22: return ( Lower-Bound < Upper-Bound );
23: end.
```

In NC*-CBJ, a global cost $C_{\emptyset}$ is maintained which is initially zero. For each domain D_j an additional set, $\hat{D}_j$, is maintained which holds the values that were removed from D_j. After every assignment, all costs of all values are updated as in update-state procedure. Then, for each future variable $j \in F$ the minimal cost of all values in D_j, c_j, is calculated (line 3). In the loop from lines (4-12), all given cost for values determined via using algorithm(3) is decreased by c_j. As a result of this substraction, the domain of every unassigned variable includes at least one value whose cost is zero. Then, for each produced value the first c_j assignments in its conflict-list are removed and added to GCS. And therefore, the GCS includes the union of the conflict sets of all assigned and unassigned variables. In line 13, a global lower bound, LB, on the number of inconsistency constraints is calculated as the sum of the current solution's cost and $C_{\varnothing}$, which is next used to prune values against the upper bound. If any lower bound of a value which is given by the sum of cost(CPS), $C_{\varnothing}$, and its own cost, exceeds the limit of the upper bound then the value is removed from the variable's domain as shown in (lines 14-21).

4. Partial Incompatible Based Lower Bound of NC*

In this section, we define partially incompatible relation over set of variables of the given CSP. Upon this relation the algorithm M-NC*-CBJ calculates a new quantity by which the lower bound of NC*-CBJ algorithm is improved. The modified bound, M-lower-bound, is based on successive checking for the validity of partial incompatibility pairs among future variables. In the following, the partial incompatible relation between two variables is defined.

Definition 4.1. *Let i and j be two variables in a CSP and let $S_k \subseteq D_k$ for $k \in \{i, j\}$. i and j are said to be partially incompatible with respect to S_i and S_j denoted by P-incomp(S_i, S_j) if and only if $\exists$ R_{ij} such that $\forall$ $a \in S_i$ and $\forall$ $b \in S_j$ we have $(a, b) \notin R_{ij}$.*

The introduced modification of NC*-CBJ function named M-NC*-CBJ, is described in Algorithm (5). In this algorithm lines (1-12) are the same as in NC*-CBJ( ) given in Algorithm (4). In the suggested algorithm, a set *W* is used to store the future variables which contribute in the new bound. *W* is testing later to modify the lower bound; at the beginning *W* is an empty set. For current variable *i*, the algorithm computes a local bound using a function Incomp-bound(*i*) which is described in details in Algorithm(6). The returned value by Incomp-bound(*i*) corresponds to the number of disjoint future variable pairs which are partially incompatible with respect of their sets of supported values. The computed value by the function Incomp-bound(*i*), is added to the previous lower bound that is computed by NC*-CBJ as stated in (line 14). In lines (15-28), the new lower bound (M-lower-bound) is used to prune values against an upper-bound. Values are pruned according to its cost. If the cost of a value *a* of future variable *j* is zero, then this value is pruned when the M-lower-boun $\geq$ Upper-Bound. Otherwise, the value is pruned if ($C_j(a)$+ M-lower-bound - (check(*j*, *W*))) $\geq$ Upper-Bound) as mentioned in line 23. The function check (*j*, *W*) returns one if the variable $j \in W$; otherwise returns zero.

Algorithm (5):

Function M- NC*- CBJ(*i*)

```
1:  begin
2:  for $j \leftarrow i + 1$ to $n - 1$ do
3:     $c_j \leftarrow$ C-cost($j$);
4:     for all $a \in (D_j \cup \hat{D}_j)$ do
5:        $C_j(a) \leftarrow C_j(a) - c_j$ ;
6:        for 1 to $c_j$ do
7:           GCS $\leftarrow$ GCS $\cup$ first-element($L_j(a)$);
8:           delete first-element($L_j(a)$);
9:        end for
10:    end for
11:    $C\emptyset \leftarrow C\emptyset + c_j$ ;
12: end for
13: W $\leftarrow \emptyset$
14: M-lower bound $\leftarrow$ cost(CPS) + $C\emptyset$ + Incomp-bound($i$) ;
15: for $j \leftarrow i + 1$ to $n - 1$ do
16:     for all $a \in D_j$ do
17:         if $C_j(a) = 0$ then
18:             if (M-Lower-Bound $\geq$ Upper-Bound) then
19:                 $\hat{D}_j \leftarrow \hat{D}_j \cup \{a\}$;
20:                 $D_j \leftarrow D_j \setminus \{a\}$;
21:             end if
22:         else
23:             if ($C_j(a)$ + M-Lower-Bound - check($j$, $W$) $\geq$ Upper-Bound) then
24:                 $\hat{D}_j \leftarrow \hat{D}_j \cup \{a\}$;
25:                 $D_j \leftarrow D_j \setminus \{a\}$;
26:             end if
27:         end if
28:     end for
29: end for
30: return ( Lower-Bound < Upper-Bound );
31: end.
```

To complete the description of Algorithm (5) we give some details on the main idea of computing the new bound, Incomp-bound(*i*) function. In Which, for each future variable *j* we add it to a set *F* which is including the future variables of the current variable. Then, construct a set S_j which contains all supported values *a*, where $a \in D_j$ such that $C_j(a) = 0$ as shown in (lines 5-10). Next, for each two different future variables *j* and *k* we check wether or not they are partially incompatible with respect to S_j and S_k. If they are partially incompatible, then the bound is increased by one. After that, *j* and *k* are added to set *W* and removed from *F* (line 12-17). Otherwise, get a new pair of variables that belongs to *F*. When there is no more partially incompatible variables among future variables, the computed value is returned by the algorithm.

Algorithm (6)
Function Incomp-bound(*i*)

```
1:  begin
2:  $F \leftarrow \emptyset$;
3:  for $j = i+1$ to $n-1$ do
4:        $F \leftarrow F \cup \{j\}$;
5:        for all $a \in D_j$ do
6:              if $C_j(a) = 0$ then
7:                  add $a$ to $S_j$ ;
8:              end if
9:        end for
10: end for
11: bound = 0;
12: for all $j, k \in$ F do
13:           if (($j \neq k$) & (P-incomp($S_j$ , $S_k$)) ) then
14:                 bound $\leftarrow$ bound + 1;
15                  W $\leftarrow$ W $\cup \{j, k\}$;
16:                 F $\leftarrow$ F \ $\{j, k\}$;
17:           end if
18: end for
19: return bound;
20: end.
```

As illustrative application for showing the improvement of a lower bound is given in the following example.

Example

Given binary CSP Z = (X, D, R) having X= {1, 2, 3, 4}, D_i = {a, b,c} for i $\in$ {1, ..., 4}, and R= { R_{12} , R_{13} , R_{14} , R_{23} , R_{24} , R_{34} }, where R_{12}= { (a, a) , (a, b), (c, a) }, R_{13}= { (a, b), (b, a) , (c, b), (b, c) }, R_{14} = { (b, c) , (b, a), (c, b) }, R_{23} = {(a, c) ,(c, b) }, R_{24} = {(b, a) ,(b, c) ,(c, a) }, R_{34} = { (b, b) ,(c, c) ,(a, b) }.

Executing the steps of algorithm(1) including modification, one can obtain the following:
- Resulting in line (1-13) is CPS = { (1, a)}, cost(CPS)=0, F= { 2, 3, 4} and cost of each value in the domain of future variables is updating as follows.

	$C_2(.)$
a	0
b	0
c	1

	$C_3(.)$
a	1
b	0
c	1

	$C_4(.)$
a	1
b	1
c	1

By applying the function M-NC*-CBJ (Algorithm(5)), we obtain $C_{\varnothing}$ = ((0)+(0)+(1))= 1. Incomp-bound(1) = 1, this due is to partially incompatible relation between variables 2, 3 with respect of their sets of supported values (these sets are ((2, {a, b}), (3, b)). M-lower-bound= cost(CPS) + $C_{\varnothing}$ + Incomp-bound(1) = 2.

It is important to point that, The M-lower-bound at (2, c) = $C_2(c)$ + M-lower-bound - check(2, W)= 1 + 2 - 1 = 2. Since variable $2 \in$ W (variable 2 partially incompatible with variable 3). Also, lower-bound at (3, a), (3, c) is equal 2.

For comparison, we repeat the same task by using the function NC*-CBJ( ). We obtain that the previous lower-bound(LB)= cost(CPS) + $C_{\varnothing}$ = 0 + 1 = 1. Evidently, the lower bound of adding the value returned by function Incomp-bound(1) is greater than the previous lower bound.

5. The Correctness of an Improved Lower Bound of M-NC*-CBJ

Now, the complexity of computing the number of disjoint future variable pairs which are partially incompatible is obtained via using two sequential steps as follows. First, we suppose *n* is the number of variables and *d* is the maximum domain size of variables .

1. For every future variable the number of required steps to search for the values with cost zero is at most *d*. Then, the total number of searching steps is *O*(*nd*).

2. For all possible $\frac{n(n-1)}{2}$ pairs of future variables, testing the partially incompatible relation between them is O($\frac{n(n-1)d^2}{2}$).

So, the required complexity is O($nd + \frac{n(n-1)d^2}{2}$) approximately $O(n^2d^2)$. It is worthy to say that this computation is a polynomial time, so the execution time of the whole algorithm is not affected by the suggested modification.

The Correctness:

Here, we show that the additional part to lower bound of NC*-CBJ algorithm assists to non-decreasing the value of lower bound, i.e., the lower bound of M-NC*-CBJ algorithm is improved since in most cases, it is increasing.

In the rest of this section, the following notations are used. F(*i*) and W_i are the set of future variables of *i* and set of variables (if exist) contribute in incomp-bound(*i*), respectively. Further, $C_{\varnothing}^{i}$ is the sum of lowest cost of the future variables domain after assigning *i*.

In the following[6], we prove that M-lower-bound; the lower bound of NC*-CBJ plus the additional part, is still lower bound. This is done via checking for every variable *i* the value of

[6] We use static ordering on the set of variables X ={1, 2, ...,n}

additional part at next variable *i+1*. The variable *i+1* belongs to W_i or not; in each cases we show that it assigned with supported or not supported value. These four cases are discussed in the next lemmas.

Lemma 5.1. *Let i be a current variable and then, assume that i+1* $\in W_i$. *Let c be a supported value assigned to i+1. For every k* $\geq$ *1 if* $C_\varnothing^{i+1} = C_\varnothing^{i} + k$, *then Incomp-bound(i+1)* $\geq$ *Incomp-bound(i)- k is satisfied.*

Proof. As a result of given $i+1 \in W_i$ and it has assigned supported value c, then there exists at least one future variable, say *l*, belonging to F(*i*+1). The cost of all supported values of *l* must increase by one (C-cost(*l*)=1). Thus, *l* is the variable which was partially incompatible with *i+1* in computing incomp-bound(*i*) as mentioned in definition (4.1). To complete the proof, we use mathematical induction on k.

The basis step at k=1 : According to $C_\varnothing^{i+1} = C_\varnothing^{i} + 1$, there exists exactly one future variable whose cost of all its supported values increased by one as shown in Algorithm (5) (lines 2-12). Obviously, this future variable is *l*. Then, for each other future variables $x \in$ F(*i*+1) other than *l*, their domains contain at least one supported value its cost not increased (C-cost(*x*)=0). So, the number of pairs of future variables partially incompatible at *i*+1 is decreased by exactly one pair than the existing in *i*. This pair is (*i*+1, *l*), since *i*+1 is deleted from calculation of incomp-bound(*i*+1). But this number is affected by new partially incompatible pairs that may appear through computation of incomp-bound(*i*+1). Hence,

Incomp-bound(*i*+1) $\geq$ Incomp-bound(*i*)- 1.

Induction Hypothesis: the statement is true at k = *m* which means that if $C_\varnothing^{i+1} = C_\varnothing^{i} + m$, then Incomp-bound(*i*+1) $\geq$ Incomp-bound(*i*)- *m*. Now, we show that the statement is true at *k*=*m*+1. Given $C_\varnothing^{i+1} = C_\varnothing^{i} + (m+1)$ means that there are m+1 future variables such that in each variable cost of all supported values is increased as given in Algorithm (3) lines (10-17). By applying induction hypothesis on *m* future variables we have Incomp-bound(*i*+1) $\geq$ Incomp-bound(*i*)- *m*. Than, adding one variable, say l_1, to the *m* variables leads to increasing cost of all l_1 supported values and also the lowest cost of l_1 is increased. So, it is possible to add more values to be supported. Then, we discuss the following two cases depending on the possibility of finding $l_1 \in W_i$:

1. $l_1 \in W_i$, this means that there exists another variable ,say l', such that $l_1 \neq l'$ and P-incomp($S_{l_1}, S_{l'}$)in computing Incomp-bound(*i*) see Algorithm (6) lines(12-18). The possibility of adding more supported values to D_{l_1} may lead to not satisfaction of P-incomp($S_{l_1}, S_{l'}$) at computing Incomp-bound(*i*+1). Hence, Incomp-bound(*i*+1)$\geq$ Incomp-bound(*i*)- *m* -1.

2. $l_1 \notin W_i$. Then, we have two subcases.
   (a) $l_1 \notin W_{i+1}$ Incomp-bound(i+1)$\geq$ Incomp-bound(i)- m.
   (b) $l_1 \in W_{i+1}$, there exists another variable ,say l_2, such that $l_1 \neq l_2$ and P-incomp(S_{l_1}, S_{l_2}) at computing Incomp-bound(i+1) as shown in algorithm (6) (lines 12-17). When $l_2 \notin W_i$, then Incomp-bound(i+1) > Incomp-bound(i)- m. Otherwise, we have Incomp-bound(i+1)$\geq$ Incomp-bound(i)- m.

From the two cases 1 and 2, we have Incomp-bound(i+1)$\geq$Incomp-bound(i)- (m+1).

Lemma 5.2. *Let i be a current variable and then, assume that i+1* $\in W_i$. *Let c be a non-supported value, i.e.,* $C_{i+1}(c) > 0$, *assigned to i+1. For every k* $\geq$ *0 if* $C_{\varnothing}^{i+1} = C_{\varnothing}^{i} + k$, *then Incomp-bound(i+1)*$\geq$ *Incomp-bound(i) - k - 1 is satisfied.*

Proof. It is easy to deduce the validity of the required statement using mathematical induction on k as was done in the proof of Lemma (5.1).

The basis step at k=0 : substituting k = 0, we have $C_{\varnothing}^{i+1} = C_{\varnothing}^{i}$. Then, for each future variables $x \in$ F(i+1), their domains contain at least one supported value and its cost not increased (C-cost(x)= 0) as given in Algorithm (3) lines (10-17). So, the number of pairs of future variables partially incompatible at *i+1* is decreased by exactly one pair than the existing number of partially incompatible pairs at *i*. Obviously, the removed pair includes *i+1*, since *i+1* is deleted from the calculation of incomp-bound(i+1). But, this number is affected by new partially incompatible pairs that possibe appeare through the computation of incomp-bound(i+1).
Hence, Incomp-bound(i+1)$\geq$ Incomp-bound(i)- 1.

It is easy to follow the same proof of Lemma (5.1) by using induction hypothesis at k = m which is if $C_{\varnothing}^{i+1} = C_{\varnothing}^{i} + m$, then Incomp-bound($i$+1)$\geq$ Incomp-bound(i) - m - 1.

Lemma 5.3. *Let i be a current variable and then, assume that i+1* $\notin W_i$. *Let c be any value belongs to* D_{i+1} *assigned to i+1. For every k*$\geq$ *0 if* $C_{\varnothing}^{i+1} = C_{\varnothing}^{i} + k$, *then Incomp-bound(i+1)*$\geq$*Incomp-bound(i)-k is satisfied.*

Proof. Similar to the above lemmas, we use mathematical induction on k to prove the statement. The basis step at $k=0$: since $C_{\varnothing}^{i+1} = C_{\varnothing}^{i}$, then for each future variable $x \in$ F(i+1), its domain contains at least one supported value whose cost doesn't increase (C-cost(x)=0). So, the number of pairs of future variables partially incompatible at i still partially incompatible at *i+1*. But this number is affected by new partially incompatible pairs that possible appear through the computing of incomp-bound(i+1). Hence, Incomp-bound(i+1)$\geq$Incomp-bound(i).

Following the same proof of Lemmas 5.1 and 5.2, using induction hypothesis at $k = m$. If $C_{\varnothing}^{i+1} = C_{\varnothing}^{i}$ + m then Incomp-bound(i+1)$\geq$ Incomp-bound(i) - m. So, the proof of this lemma is a straightforward.

Theorem 5.4. *M-Lower-Bound produced by M-NC*-CBJ algorithm is a valid lower bound.*

Proof. In order to prove that M-Lower-Bound is a lower bound, it should be proven that for each variable *i*, the value computed by

$$\text{M-LB} = \text{cost(CSP)} + C_{\varnothing} + \text{Incomp-bound}(i) \qquad (1)$$

is lower bound. It is sufficient to show that for a current variable *i*, the following two statements are satisfied:

1. The value computed by eq (1) records different inconsistencies, i.e., no inconsistency recorded twice.
2. The number of inconsistencies of any node descending from *i* is greater than or equal The number of inconsistencies at *i*.

First, no inconsistency computed by eq (1) was recorded twice due to the following:

- The value cost(CPS) records inconsistencies between only past variables according to Definition 2.1.
- The value $C_{\varnothing}$ records inconsistencies between past and future variables as mentioned in Algorithm(5) (lines 2-12).
- The value Incomp-bound(*i*) records inconsistencies between future variables only as a result of Algorithm (6) lines (12-18). As given in line 16 of algorithm (6), no inconsistency between future variables was recorded twice since any future variable contributes in computing Incomp-bound(*i*) by at most one.

Secondly, using Lemma (5.1-5.3) and the mathematical induction on the depth of descending nodes from i , it is easy to show that the number of inconsistencies of any node descending from *i* is greater than or equal the number of inconsistencies computed by M-Lower-Bound at *i*. This leads to proving the validity of statement 2.

6. Experimental Results

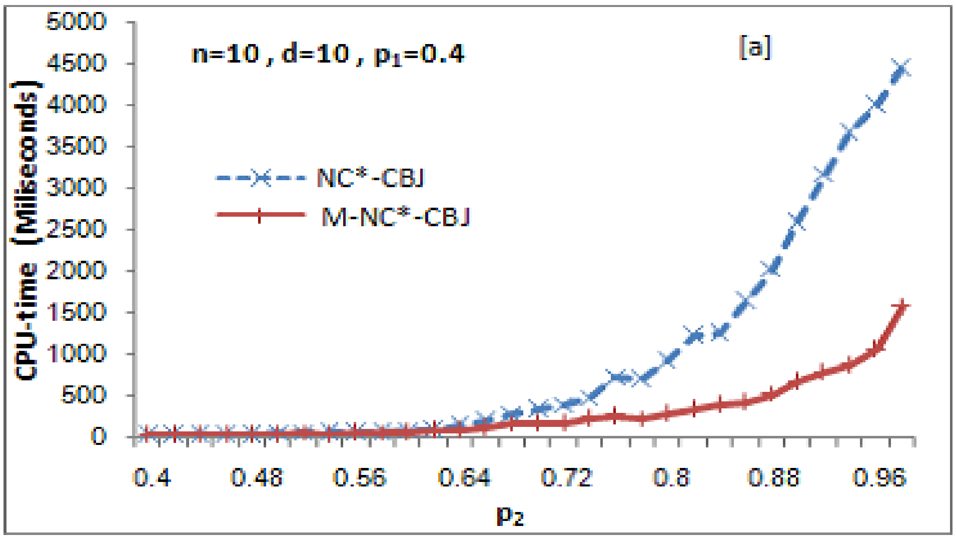
In this section, we evaluate the contribution of the new lower bound, M-lower-bound, on improving the previous one given in [7]. The performances of the two algorithms M-NC*-CBJ and NC*-CBJ on random binary CSPs are compared. Every given binary random CSPs are characterized by (n, d, p_1, p_2), where n is the number of variables; d is the number of values per variable; p_1 is the graph connectivity (the ratio of existing constraints); p_2 is the constraint tightness (the ratio of forbidden value pairs). The constrained variables and the forbidden value pairs are randomly selected [9]. The following problem classes have been experimented:

1. (10, 10, 0.4, p_2), where p_2 starts with 0.4 and increases by 0.02 in each run until 0.98
2. (10, 10, 0.6, p_2), where p_2 starts with 0.4 and increases by 0.02 in each run until 0.8.
3. (10, 10, 0.8, p_2), where p_2 starts with 0.4 and increases by 0.02 in each run until 0.7.

For each problem class and each parameter setting, samples of 50 instances are generated. Both algorithms are implemented in C++ and run on PC (2.0 GHZ, Pentium IV). Each problem class is solved by NC*-CBJ and M-NC*-CBJ. The two algorithms use static variable ordering which is decreasing forward degree, breaking ties with decreasing backward degree as mentioned in [3]. Values are always selected by increasing cost. We draw the curves between the execution time and total number of assignments against p_2, respectively, to compare the search efforts of M-NC*-CBJ and NC*-CBJ in the three tested classes, as illustrated in Figures (1-3). As it can be observed, M-NC*-CBJ improves practically NC*-CBJ in all problem classes.

Figure 1(a) illustrates the comparison between the execution of M-NC*-CBJ and NC*-CBJ in case of the relation between p_2 and the CPU-time in milliseconds at p_1=0.4. Clearly, its shown that M-NC*-CBJ outperforms NC*-CBJ by a ratio within 0.64 to 4.18. At the same time, we compare between the same algorithms at $p_1 = 0.4$ in case of the relation between p_2 and number of assignments as given in Figure 1(b). Also, the performance of M-NC*-CBJ is better than NC*-CBJ by a ratio within 0.94 to 2.98.

Similarly, Figure 2 shows the same comparisons at p_1 = 0.6. Figure 2(a) show that M-NC*-CBJ outperforms NC*-CBJ by a ratio within 0.81 to 3.51. Figure 2(b) show that M-NC*-CBJ outperforms NC*-CBJ by a ratio within 1.03 to 2.49.



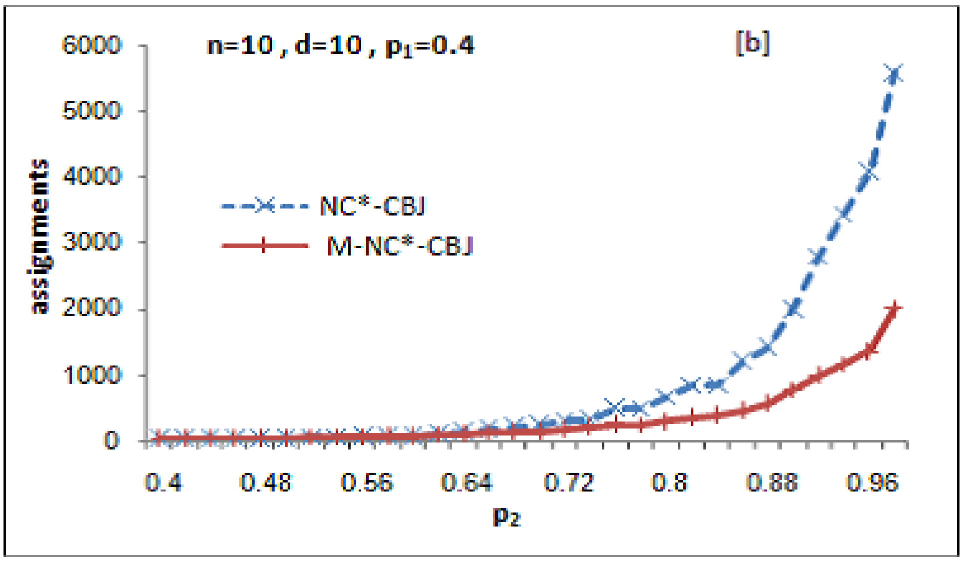
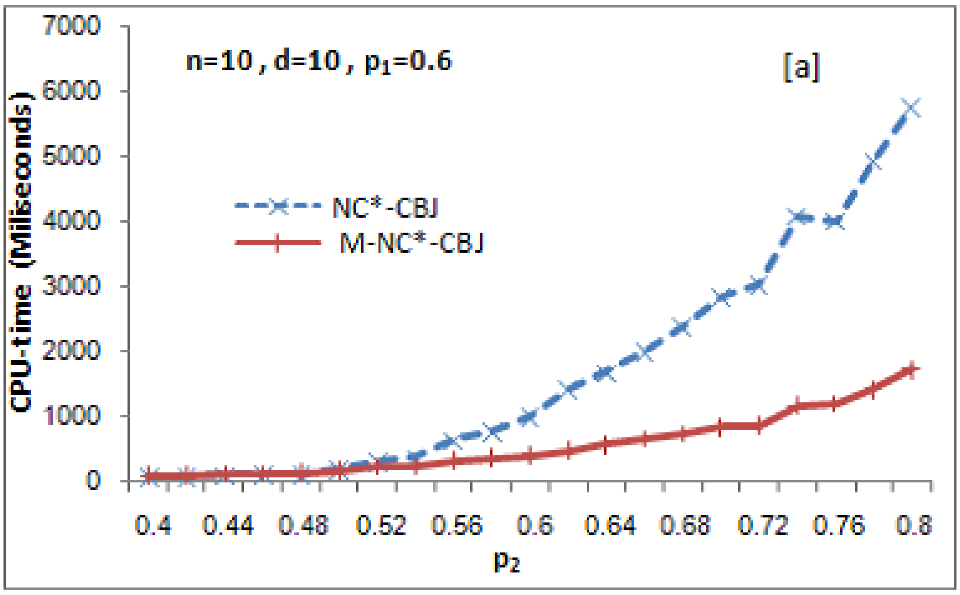


Figure 1: (a) The relation between CPU-time and p_2 when $p_1 = 0.4$; (b) The relation between the number of assignments and p_2 when $p_1 = 0.4$



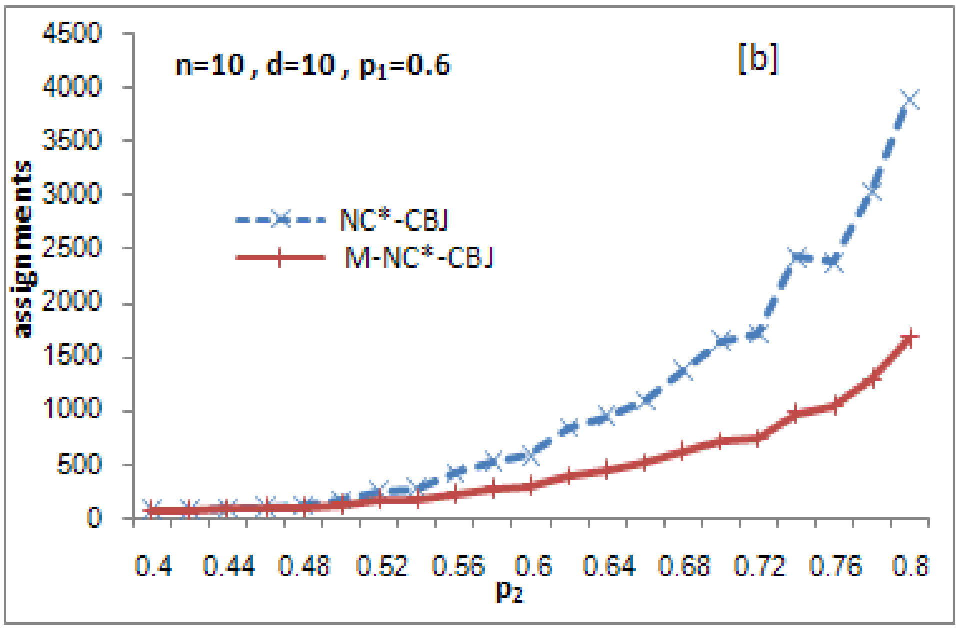
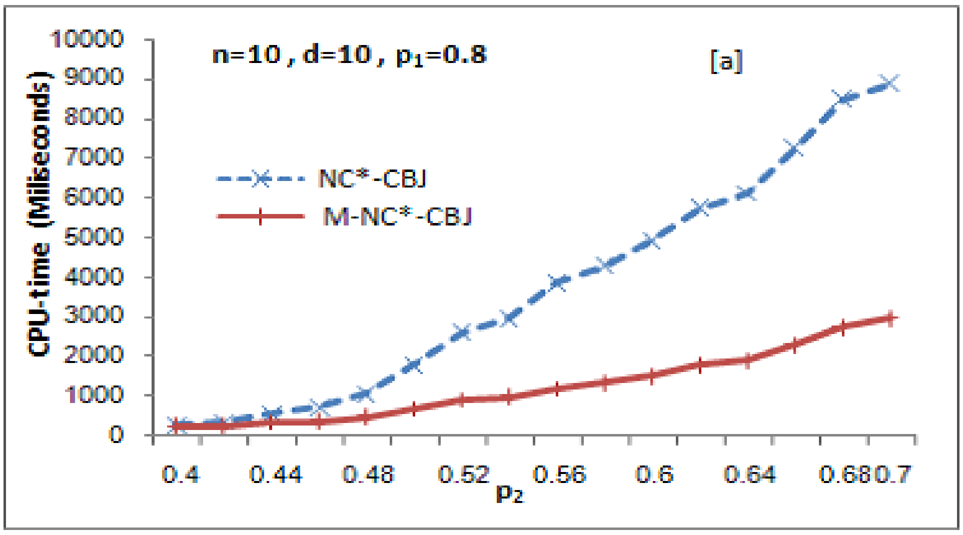


Figure 2: (a) The relation between CPU-time and p_2 when p_1 =0.6; (b) The relation between the number of assignments and p_2 when p_1 =0.6



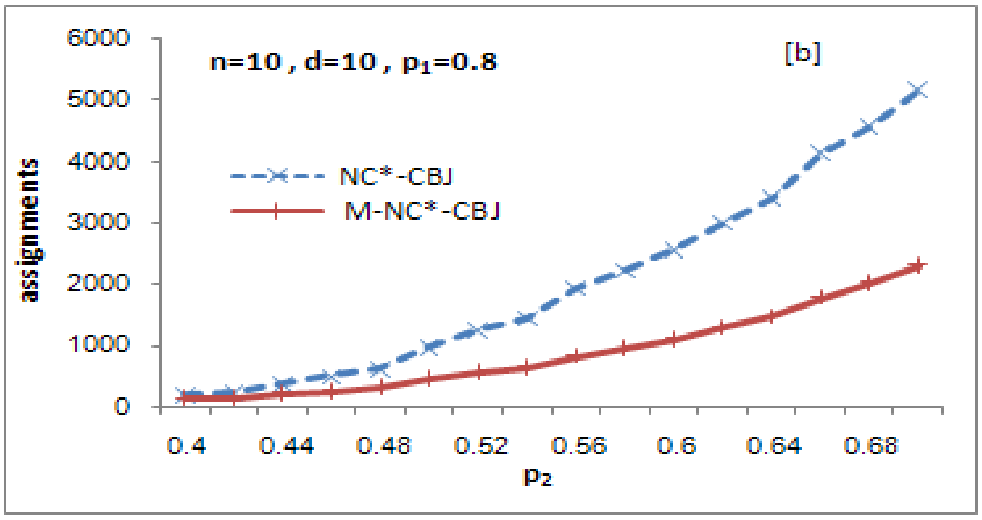


Figure 3: (a) The relation between CPU-time and p_2 when p_1 =0.8; (b) The relation between the number of assignments and p_2 when p_1 =0.8

Finally, Figure 3 illustrates the same comparison at $p_1 = 0.8$. Figure 3 (a) shows the curves of the relation between p_2 and CPU-time in millisecond. Evidentely, from figure 3(a), the performance of M-NC*-CBJ is much better than NC*-CBJ with a ratio within 1.23 to 3.29. In addition, figure 3(b) gives the same conclusion when drawing the curves between p_2 and the number of assignments i.e., the ratio of the perofrmance between the two algorithms varies within 1.33 to 2.33.

Through the curves of our experimental results once can observe that the improvement factor is increasing with the increase in constraint tightness (p_2).

7. Conclusion and Future Work

Branch and Bound (BB) is a general algorithm for finding optimal solutions of various optimization problems, especially Max-CSPs. The efficiency of this algorithm depends critically on the effectiveness of the lower bound. Any increment in the lower bound causes large gains in the search space reduction. As a result of this reduction the required solution is obtained more quickly. In this paper, we give a new bound resulting from the proposed partially incompatible relation between future variables. Adding this bound to the lower bound of NC*-CBJ algorithm leads to a new improving lower bound denoted by M-Lower-Bound. Then, we prove the validity of M-Lower-Bound and demonstrate its effect on the M-NC*-CBJ algorithm.

Furthermore, the experimental results affirmed that the efficiency and performance of M-NC*-CBJ algorithm are better than the previous NC*-CBJ algorithm. According to the paper's results, we conclude that the lower bound quality remains the major issue in algorithms for solving Max-CSPs.

In future work, we suggest applying the introduced modification of lower bound to other advanced consistency properties such as AC* and FDAC.

References

[1] S. Bistarelli, P. Codognet, Y. Georget, and F. Rossi, "Labeling and Partial Local Consistency for Soft Constraint Programming", second international workshop on Practical Aspects of Declarative Languages, PADL00, p. 230-248, 2000.

[2] E. Freuder, and R. Wallace, "Partial Constraint Satisfaction", Artificial Intelligence, vol. 58, pp. 21-70, 1992.

[3] J. Larrosa and P. Meseguer,"Exploiting the Use of DAC in MAX-CSP", Proceedings of CP-96, pp. 308-322, 1996.

[4] J. Larrosa and P. Meseguer, "Partition-based lower bound for Max-CSP", Proceedings of the 5th International Conference on Principles and Practice of Constraint Programming (CP-99), pp. 303-315, 1999.

[5] J. Larrosa, P. Meseguer, and T. Schiex, "Maintaining reversible DAC for Max-CSP", Artificial Intelligence, vol. 107, pp. 149-163, 1999.

[6] J. Larrosa, "Node and Arc Consistency in Weighted CSP", Proceedings of the Eighteenth National Conference on Artificial Intelligence (AAAI-2002), pp. 48-53, 2002.

[7] J. Larrosa and T. Schiex, "Solving weighted csp by maintaining arc consistency" , Artificial Intelligence, vol. 159, pp. 1-26, 2004.

[8] P. Prosser, "Binary constraint satisfaction problems: some are harder than others", Proceedings of ECAI-94, pp. 95-99, 1994.

[9] Random Uniform CSP Generators, http:// www.lirmm.fr/ bessiere/ generator.html, 1996.

[10] J.C. R´egin, T. Petit, C. Bessi´ere, and J-F. Puget, "New lower bounds of constraint violations for over constrained problems", Proceedings of CP, pp. 332-345, 2001.

[11] T. Schiex, "Arc consistency for soft constraints", Proceedings of CP- 2000 Singapore, pp. 411-424, 2000.

[12] T. Schiex, "Une comparaison des coherences darc dans les Max-CSP", Proceedings of JNPC, pp. 209-223, 2002.

[13] E. Tsang, "Foundations of Constraint Satisfaction", Academic Press, 1993 .

[14] R. J. Wallace and E. Freuder, "Conjunctive width heuristics for maximal constraint satisfaction", Proceedings of AAAI-93, pp. 762-768, 1993.

[15] R. J. Wallace, "Directed arc Consistency preprocessing", Proceedings of the ECAI-94 Workshop on Constraint Processing (LNCS 923), pp. 121-137, 1994 .

[16] R. J. Wallace, "Enhancements of Branch and Bound Methods for the Maximal Constraint Satisfaction Problem", Proceedings of AAAI-96, pp. 188-195, 1996.

[17] R. Zivan and A . Meisels, "Conflict directed Backjumping for Max-CSPs", Proceedings of IJCAI07, pp. 198-204, 2007.