Dynamic Constraint Ordering Heuristics
for Dynamic CSPs

Saida Hammoujan*, Yosra Acodad*
Imade Benelallam* ‡ and El Houssine Bouyakhf*

* LIMIARF, Faculty of Sciences, Mohammed V University - Agdal, Rabat, Morocco
‡ National Institute of Statistics and Applied Economics - IRFANE, Rabat, Morocco

hammoujan@gmail.com, yosra.acodad@gmail.com
imade.benelallam@ieee.org, bouyakhf@fsr.ac.ma

Copyright © 2013 Saida Hammoujan et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Abstract

Many research studies have shown that the choice of a relevant order of variables and/or values can significantly improve the efficiency of search algorithms. However, in Constraint Satisfaction Problems CSPs, constraint ordering heuristics have received less attention than variables and values ordering heuristics. In the literature, only some few contributions have been done on this area, and mainly as a preprocessing phase before starting the search process. In this paper, we show that constraint ordering heuristics are promising and able to make relevant choices during the search process within a Dynamic CSP.

Our proposed approach, called Dynamic Constraint Ordering heuristic DCO, selects dynamically constraints which are likely to lead to conflicts. So, we apply a ”fail first” principle, in a manner that a dynamic intelligent backtracking event can be performed earlier. This method is able to guide a dynamic repair algorithm towards the hardest constraint subproblems and to tackle inconsistencies in order to correct solution assignment. The conducted experiments have confirmed that the proposed approach is interesting. Our results show clearly the efficiency of the heuristics combined with Partial-Order Dynamic Backtracking.
algorithm PDB on Meeting Scheduling Problems. The algorithms have been tested in terms of run-time, constraints checks and solution stability.

**Keywords:** Dynamic Constraint Satisfaction Problems, Partial-order Dynamic Backtracking, Constraint Ordering Heuristics, Repair solution

1 Introduction

Constraint Satisfaction Problem (CSP) formalism offers a simple, natural and powerful knowledge representation framework and various algorithms for solving Artificial Intelligence related problems. It consists of a set of variables, each associated with a domain of values, and a set of constraints. Each of the constraints is expressed as a relation, defined on some subset of the variables, denoting the consistent value assignments that satisfy the constraint. A solution to a CSP is an assignment of a value to every variable, in such a way that every constraint is satisfied. Nevertheless, many problems in AI require reasoning in dynamic environments, which the knowledge is not defined once and for all, but may incrementally evolve between successive requests, and the old solution of this Dynamic Constraint Satisfaction Problem (DCSP) has to be corrected according to the environment behaviour (changes). A Dynamic Constraint Satisfaction Problem (DCSP) \[12\] is an extension to a static CSP that models addition and retraction of constraints and hence it is more appropriate for handling dynamic real-world problems.

In the literature, several techniques has been developed to increase the efficiency of the dynamic constraint satisfaction algorithms. Some of these techniques are based on a sequence of decision to make, when solving a CSP using backtracking search, as which variable to instantiate next and which value to choose first. These decisions are referred to as the variable and value ordering decisions. It has been shown that for many problems, the best choice of variable and value ordering can have a substantial impact on the performance to find a solution when one exists \[11\]. However, only some few works have been done on constraint ordering heuristics, but mainly as a preprocessing step to reduce redundancy in constraint networks \[8\].

The problem considered here is to find a new solution of a dynamic constraint network after each modification of the original problem. The present paper focuses on scenarios where after a CSP is solved, some of the constraints have changed and the solution to the newly generated CSP is required to consider all these perturbations. According to \[4\], we consider that all possible changes to a CSP (constraint or domain modifications, variable additions or removals) can be expressed in terms of constraint additions, removals or changes.
In this paper, we present a new constraint ordering approach for repairing solutions in dynamic CSPs, called Dynamic Constraint Ordering (DCO) heuristic. The proposed heuristic selects dynamically the constraints which is likely to lead to a conflict as soon as possible. This technique guides the repair algorithm, Partial-Order Dynamic Backtracking (PDB) [10], towards a hard constraint network sub-network in order to correct solution's assignments. The key idea of this paper is to exploit the flexibility of PDB algorithm by proposing a new dynamic constraint ordering heuristic that can be used in the context of Dynamic CSP. To the best of our knowledge, the concept of constraint ordering heuristics has never been used in such a context.

This paper is organized as follows: First, we present related work and Dynamic CSP formalism. Then, we present a description of Partial Order Dynamic Backtracking algorithm. Next, we present the principle idea of dynamic constraint ordering heuristics in PDB algorithm. After that, we evaluate the performance of our approach on randomly generated Meeting Scheduling Problems. Finally, we extract some conclusions and directions for further research.

2 Related work

In the literature, several algorithms have been developed for Dynamic Constraint Satisfaction Problems. In particular, in [4] a description of a complete algorithm (Local Changes) dedicated to the maintenance of solutions in dynamic problems has been proposed. Then different extensions [18] were proposed to LC algorithm to the management of fuzzy constraints (max aggregation operator), resulting in the FLC algorithm (Flexible Local Changes). Other authors also tried to solve the same problem using different approaches. Specifically, in [19] a tool which is dedicated to continuous planning/scheduling with interleaving between planning/scheduling and execution and uses an iterative repair approach was presented. The first reproach of these contributions, is that the authors were not interested to study the scalability of the proposed algorithms compared with the incremental changes of the resolved problems. And the second one is that no authors were interested to the study and the use of heuristic knowledge of such a dynamic constraint network.

On the other hand, the experiments and analyses conducted by several researchers have shown that the order in which variables and values are chosen during the search may have substantial impact on the complexity of the explored search space. However, we note that constraint ordering heuristics have received less attention by researchers in the literature. In this section, we present most relevant constraint ordering related work.

In [15], authors have proven that ordering heuristics can increase the efficiency of relaxation, reducing the number of value pair checks, in comparison with reference ordering. The proposed heuristics order the list of variable pairs
(constraints) so that the major effect is to minimize list additions. Authors have concluded that ordering by increasing size of the domain relaxed against is an effective general strategy for enhancing performance of algorithms that establish full arc consistency. However, in [6], authors have proposed a new constraint ordering heuristics in $AC^3$, where the set of choices is composed of the arcs in the current set maintained by $AC^3$. They used constrainedness as a heuristic for reducing the number of checks needed to establish arc consistency in $AC^3$. In [7] and [8], authors have presented a preprocess heuristic called Constraint Ordering Heuristics ($COH$) that studies the constrainedness of the scheduling problem and mainly classifies the constraints so that the tightest ones are studied first. Thus, a backtracking search algorithm can tackle the hardest part of a search problem first and inconsistencies can be found earlier.

As we have already mentioned, constraint ordering has received less attention and there is only some few works that have been done. The mainly studied techniques are used as a preprocessing approach in constraint propagation to achieve arc-consistency or as constraint heuristics for reducing redundancy in constraint networks. As far as the authors are aware, there exists no work related to the use of constraint ordering heuristics in the context of $DCSP$.

3 Background

A Constraint Satisfaction Problem (CSP) involves a finite set of variables, each taking values in a finite domain, and a finite set of constraints over variables, that forbids/allows some combinations of assignments on these variables [1]. Formally speaking, a CSP can be described such that:

Definition 3.1 (CSP) A Constraint Satisfaction Problem is a tuple $(X,D,C)$:
- $X$ is a set containing $n$ variables $\{x_1, \ldots, x_n\}$;
- $D$ is a set of domains $\{D(x_1), \ldots, D(x_n)\}$ for these variables, with each $D(x_i)$ containing the possible values which $x_i$ may take;
- $C$ is a set of $m$ constraints $\{c_1, \ldots, c_m\}$ between variables in subsets of $X$. Each $c_i \in C$ expresses a relation defining which variable assignment combinations are prohibited/allowed for the variables in the scope of the constraint.

A partial assignment is a set of tuple pairs, each tuple consisting of an instantiated variable and the value that is assigned to it in the current search node. A full assignment is one containing all $n$ variables. A solution to a CSP is a full assignment such that no constraint is violated.

Some assignments can never be contained in a solution. These conflicts set are commonly called nogoods.

Definition 3.2 (Nogood) A nogood is an expression of the form: $(x_1 = v_1) \land \ldots \land (x_k = v_k) \rightarrow x \neq v$
A *nogood* is a failure explanation, it can be used to represent a constraint as an implication. It’s logically equivalent to the constraint:

$$\neg [(x_1 = v_1) \land \ldots \land (x_k = v_k) \land (x = v)]$$

There are clearly many different ways of representing a given constraint as a *nogood*. One special *nogood* is the empty *nogood*, which is tautologically false. The empty *nogood* follows that no solution exists for the problem being attempted.

The typical way in which new *nogoods* are obtained is by resolving together old ones. As an example, suppose we have derived the following:

$$(x_1 = a) \land (x_2 = b) \to x_4 \neq b \quad (x_1 = a) \land (x_3 = c) \to x_4 \neq c$$

$$(x_2 = b) \to x_4 \neq a$$

Where a, b and c are the only values in the domain of $x_4$. It follows that we can combine these *nogoods* to conclude that there is no solution with $x_1 = a$, $x_2 = b$ and $x_3 = c$: $\neg [(x_1 = a) \land (x_2 = b) \land (x_3 = c)]$. Moving $x_3$ to the conclusion: $(x_1 = a) \land (x_2 = b) \to x_3 \neq c$.

In practice, a CSP problem definition may change over time because of the environment or the user’s preference. Each separate change may result in a new CSP. The sequence of such CSPs is denoted Dynamic CSP.

In this dynamic evolution, a *DCSP* should face to set of changes [4, 5]. These changes can be a:

- Restriction: new constraints are imposed on a pair of variables.
- Relaxation: constraints that were present in the CSP are removed because they are no longer interesting or because the current CSP has no solution.

**Definition 3.3 (DCSP)** Consider $P$ a dynamic constraint satisfaction problem. $P$ is a sequence of static CSPs $\{P_0, \ldots, P_\alpha, P_{\alpha+1}, \ldots\}$ where each $P_i$ differs from the previous one by the addition or removal of some constraints.

In this paper we assume that all possible changes of a *DCSP* can be expressed in terms of constraint additions or removals.

On of the most important concept in dynamic environments is the study of its robustness. A functional system must persist, remain running and maintain their main features despite continuous perturbations, changes, incidences or aggressions. Thus, robustness is a concept related to the persistence of the algorithms, of its functionality and its solution quality, against external interference.

The solution quality of an algorithm can be measured by the concept of stability. A solution (meaning an equilibrium state) of a dynamical system is said to be stable if small perturbations to the solution result in a new solution that stays ”close” to the original solution for all time. Perturbations can be viewed as small differences effected in the actual state of the system [14, 16].
Definition 3.4 (Stability) A solution is stable if small modifications of the constraint set require a new solution (new consistent variable assignment) that remains close (neighbourhood) to the original solution.

The neighbourhood of solutions can be formally defined in terms of norms on n-dimensional spaces. On non metric domains (like non-ordered sets of values) the Hamming distance (H) can be applied. This n-dimensional norm measures the number of variables that have different values in $S$ and $S'$:

$$\| S' - S \| = \sum_{i=1}^{n} \rho_i H(x'_i, x_i)$$

where $H(x'_i, x_i)$ is equal to 0 iff $x'_i = x_i$, and 1 otherwise, with a optional weighted factor $\rho_i$ for each $x_i$. This criteria evaluates the number of variables that change their values. These measures evaluate the closeness of solutions in the state space.

4 Partial-order Dynamic Backtracking Approach for DCSP:

In this section, we introduce our modified version of Partial-order Dynamic Backtracking algorithm PDB [10]. We have decided to revise this approach in order to be completely adapted to the context of repairing solution in DCSP.

The main idea of the original PDB is: to start with a complete and inconsistent assignment, then, from the set of constraints which need to be satisfied we check the consistency of every constraint. For two given variables from a selected constraint, the order is not defined at the beginning, but there exists a partial order that is dynamic and partially modified during the search process using the concept of nogood introduced with Dynamic Backtracking algorithm (DBT) [9]. By analysing this approach, it was indeed easy to see that the degree of flexibility (i.e. the average number of actions that do not have ordering constraints with other actions) is very promising. And that the question, "How can we exploit this flexibility?" has been remained open till our recent contribution on the Dynamic Variable Ordering heuristics [17].

The figure 1 presents a pseudo-code of the revised version of PDB that allows us to exploit this flexibility in terms of dynamic constraint ordering: In order to reuse former solution and past reasoning techniques, we start the search with a CSP problem that contains a full assignment (e.g. past solution) and an associated retroactive data structure that consists of a set of nogoods $NgList$, saved during the past search process of PDB algorithm. Then we update the set of constraints which need to be satisfied and we check the consistency of every constraint (lines 1 to 7). If an inconsistency is detected, a new nogood is generated using the procedure $GenerateNg(listVar)$ (lines
Next, we add the new nogood into the \texttt{NgList} and we call the recursive function \texttt{RepairSolution(Rhs(ng), CL)} trying to find a new assignment to the variable \texttt{Rhs(ng)} (lines 10-11). If the constraint \texttt{c} is consistent, it is removed from the constraints set \texttt{CL} (line 13). If all constraints are satisfied, then the solution is found and the search is ended (lines 14-15).

\begin{verbatim}
Procedure PDB(CSP, newConstraint)  
1. CSP.update(newConstraint);  
2. CL ← newConstraint;  
3. solution ← False;  
4. emptyNg ← False;  
5. while (¬solution && ¬emptyNg) do  
6. for each (\texttt{c} \in CL) do  
7. if (¬\texttt{c}.isConsistant()) then  
8. listVar ← c.getListOfVar();  
9. ng ← GenerateNg(listVar);  
10. CSP.NgList.add(ng);  
11. RepairSolution(ng.getRhs(), CL);  
12. break;  
13. else CL.remove(\texttt{c});  
14. solution ← True;  
15. CSP.ReturnResult();

Function RepairSolution(x_i, CL)  
16. if (\texttt{D(x_i).isEmpty()}) then  
17. ng ← NgResolution(x_i);  
18. if (ng.isEmpty()) then  
19. emptyNg ← True;  
20. return False;  
21. if (¬RepairSolution(Rhs(ng), CL)) then  
22. return False;  
23. CSP.NgList.removeWithLhs(x_i);  
24. x_i.ChangeValue(); CL.addWith(x_i);  
25. return True;

Figure 1: Description of algorithm PDB
\end{verbatim}

As we have said, the procedure \texttt{RepairSolution(x_i, C)} tries to find an instantiation to the variable \texttt{x_i}, so we check if the domain of this variable is empty or not (line 16). If \texttt{RepairSolution(x_i, C)} cannot assign a value to the variable \texttt{x_i} (e.g. domain wipe-out), it builds a nogood \texttt{ng} that is the result of the nogood resolution of all nogoods where \texttt{x_i} appears in the right hand side of the resolved nogood \texttt{Rhs(ng)} using the function \texttt{NgResolution(x_i)} (lines 16-17). If the resolved nogood is empty, it means that no solution can be found and the search is ended (lines 18 to 20), else the function \texttt{RepairSolution(Rhs(ng), C)} is called recursively and tries to find a new instantiation to the variable \texttt{Rhs(ng)}. If the variable \texttt{Rhs(ng)} cannot change its value due to a dead-end, an empty no-
good is generated, then we return False (no solution, lines 21-22). Otherwise, \textit{RepairSolution}(x_i, C) deletes all nogoods with \(x_i\) in the antecedents, then it gives a new value to \(x_i\) from its valid domain and the set of constraints \(C\) is updated by adding the related constraints to \(x_i\) because its new assignment could make some violations (lines 23 to 26). In the next section we introduce the concept of Dynamic Constraint Ordering.

5 Dynamic Constraint Ordering:

A search algorithm for constraint satisfaction requires the order in which variables are to be considered as well as the order in which the values are assigned to the variable. Experiments and analysis of several researchers have shown that the order in which variables are chosen for instantiation can have substantial impact on the complexity of a backtrack search. Once the decision is made to instantiate a variable, it may have several values available. Again, the order in which these values are considered can have substantial impact on the performance to find the first solution. However, constraint ordering technique has received less attention than variable and value ordering, that is, the order of constraints does not really matter; all that matters is that the conjunction of constraints has to be satisfied.

The aim of this paper is to propose a stable and robust repair algorithm, in a manner that old solutions of dynamic CSPs can be corrected easily and closely using PDB algorithm. So, a disordered set of violated constraints has to be satisfied. In this situation, choosing a good order of constraints can make a significant difference in the overall effort required to repair a solution. Hence, we propose dynamic constraint-based heuristics for repairing solution in dynamic CSPs, called Dynamic Constraint Ordering DCO. These heuristics help PDB to make decision and select the most relevant constraint to be satisfied with priority.

DCO is not feasible for all search algorithms, e.g., with simple backtracking, the search is done by selecting a variable \(x_i\) as a current variable and instantiating it with a value. However, with Partial-order Dynamic Backtracking, the current state includes the list of new injected or disturbed constraints that need to be satisfied by checking the consistency of every constraint \(c_i\). Our proposed heuristics dynamically reorder the constraints that have to be satisfied in priority. So, the order in which constraints are made may have a profound effect on the search effort.

**Definition 5.1 (DCO)** Given \((c_1, \ldots, c_k)\) a list of inconsistent constraints \((CL)\) to be satisfied. Each time we need to check the consistency of a constraint, we classify \(CL\) using one of the proposed heuristics \((c_{ord1} < \ldots < c_{ordk})\) so that constraint \(C_{ord1}\) is studied first.
we denote by $<$ an order between constraints of $CL$ in increasing or decreasing order.

As we said, a constraint $c$ expresses a relation defining which variable assignment combinations are prohibited/allowed for the variables in the scope of the constraint. In this paper we focus in binary constraint which there are just two variables involved.

Before defining the different constraint ordering heuristics, we introduce initially the following definitions:
- **tightness($C_{ij}$)**, ratio between the total number of eliminated tuples to the number of total tuples.
- **weight($C_{ij}$)**, weighting constraints by introducing a counter associated with each constraint accounting the number of times the propagation of constraint led to a failure (conflict).
- **dom($x_i$)**, current domain, is the number of remaining values in the domain of $x_i$.
- **deg($x_i$)**, degree, is the number of constraints in which the variable $x_i$ is involved.
- **ddeg($x_i$)**, dynamic degree, is the number of variables that are involved in the constraints with unassigned variables.

The order of the constraints in a dynamic search is used to choose what constraint to check at each step of the search. These heuristics should be designed based on the topology of the constraint graph and weight constraints to minimize the length or the depth of each branch.

The proposed constraint ordering heuristics are as follows:
- **Ctight($C_{ij}$)** = max(tightness($C_{ij}$)), constraint tightness, selects the constraint with the biggest tightness.
- **Cweight($C_{ij}$)** = max(weight($C_{ij}$)), constraint maximum weight, selects the next constraint with maximum weight.
- **Cdom($C_{ij}$)** = min(dom($x_i$),dom($x_j$)), constraint minimum domain, selects the next constraint that has the least remaining values in domains of one of its variables.
- **Cdeg($C_{ij}$)** = max(deg($x_i$),deg($x_j$)), constraint maximum degree, chooses the constraint that one of its variables have the maximum degree.
- **Cddeg($C_{ij}$)** = max(ddeg($x_i$),ddeg($x_j$)), constraint maximum dynamic degree, chooses the constraint that one of its variables have the maximum degree.

Figure 2 shows an example of injected/modified constraints in a natural order (a) and classified using constraint tightnesses Ctight($C_{ij}$) (b).
Figure 2: (a) Non-ordered constraints (b) Ordered constraints

5.1 Description

The figure 3 presents a pseudo-code of $PDB$ using $DCO$. Before starting the search, we update the set of constraints which need to be satisfied and we call the function $ReorderCL(C,\text{heuristic})$ to reoder the constraint list $CL$ using the selected heuristic, then we check the consistency of every constraint (lines 1-3).

As it was explained in the previous section, we try to find a new value to the variable $x_i$ by using the procedure $RepairSolution(x_i,CL,\text{heuristic})$. If this procedure succeeds to assign a value to the variable $x_i$, all nogoods with the old value of $x_i$ in the antecedents are deleted, then the set of constraints $CL$ is reordered after adding the constraints related with $x_i$ using the selected heuristic (lines 23 to 26).
Dynamic constraint ordering heuristics for dynamic CSPs

Procedure PDB\textsubscript{DCO}(CSP, newConstraint, heuristic)\textsuperscript{*}
1. CSP.update(newConstraint);
2. CL ← newConstraint;
3. ReorderCL(C, heuristic);\textsuperscript{*}
... 
11. RepairSolution(ng.getRhs(), CL, heuristic);\textsuperscript{*}
12. break;
13. else CL.remove(c_i);
14. solution ← True;
15. CSP.ReturnResult();

Function RepairSolution(x_i, CL, heuristics)
... 
21. if (¬ RepairSolution(Rhs(ng), CL, heuristic))\textsuperscript{*} then
22. return False;
23. CSP.NgList.removeWithLhs(x_i);
24. x_i.ChangeValue();
25. CL.addWith(x_i);
26. ReorderCL(CL, heuristic);\textsuperscript{*}
27. return True;

Figure 3: Description of algorithm PDB\textsubscript{DCO}

The following example shows the difference between the PDB and the PDB\textsubscript{DCO} heuristic.

Figure 4: (a) Original problem (b) Disturbed problem

Suppose that the original problem (figure 4 (a)) includes:
• Four variables: X = \{X_1, X_2, X_3, X_4\}
• Domain of each variable is: D_1=D_2=[1,2], D_3=[1,2,3], D_4=[1,2,3,4]
• Inequality constraints: C_1: between [X_1, X_2], C_3: between [X_3, X_4]
• Equality constraints: C_2: between [X_1, X_3], C_4: between [X_2, X_4].
Using any search algorithm, the solution of this problem is:

\[ X_1 = 1, \ X_2 = 2, \ X_3 = 1, \ X_4 = 2 \]

Suppose that after finding this solution, an unexpected change in the environment (figure 4(b)) arises due to the addition of an inferiority constraint: \( C_5 \): between \([X_2, X_3]\) and the perturbation of \( C_2 \) and \( C_4 \) (inequality). In this case, the first solution will not be more valid.

In table 1, \( PDB \) is executed. The initial assignment violates three constraints \( C_2, C_4 \) and \( C_5 \). We assume that the Constraints List \( CL \) to check is: \([C_2, C_4, C_5] \) (lexicographic order). The first selected constraint from the list is \( C_2 \). This constraint is violated so it generates a nogood and \( X_3 \) is selected to change its current value and choose another value from its current domain. This new assignment could violate the linked constraints to \( X_3 \). In this case we add all these constraints \([C_3, C_5]\) to \( CL \) to check them. After \( X_3 \) chooses its value, the constraint \( C_2 \) is checked again. This time, \( C_2 \) is consistent and it is removed from \( CL \). The next checked and violated constraint is \( C_4 \) then \( X_4 \) changes its value and its linked constraint \([C_3]\) is added to \( CL \). Since the new value of \( x_4 \) is consistent, we check the next constraint \( C_5 \). As this constraint is inconsistent, \( X_3 \) is selected to change its current value and \([C_2, C_3]\) are added to \( CL \). After \( X_3 \) chooses its value, the constraint \( C_5 \) is checked again. This time, \( C_5 \) is consistent and it is removed from \( CL \). The current \( CL \) to check is: \([C_3, C_2]\). Remaining constraints in \( CL \) are satisfied, then the solution is found.

In table 2, \( PDB\_DCO \) is executed on the same problem and the selected heuristic is \( Ctight \).
The prohibited tuples and the tightness for every constraint are:

- \( C_1 = \{(1,1)(2,2)\} \) \( \Rightarrow \) \( C_{\text{tight}}(C_1) = 2/4 = 0.50 \)
- \( C_2 = \{(1,1)(2,2)\} \) \( \Rightarrow \) \( C_{\text{tight}}(C_2) = 2/6 = 0.33 \)
- \( C_3 = \{(1,1)(2,2)(3,3)\} \) \( \Rightarrow \) \( C_{\text{tight}}(C_3) = 3/12 = 0.25 \)
- \( C_4 = \{(1,1)(2,2)\} \) \( \Rightarrow \) \( C_{\text{tight}}(C_4) = 2/8 = 0.25 \)
- \( C_5 = \{(1,1)(2,1)(2,2)\} \) \( \Rightarrow \) \( C_{\text{tight}}(C_5) = 3/6 = 0.50 \)

We start by reordering the list of constraints \( CL \) using \( C_{\text{tight}} \) heuristic. This heuristic classifies the constraints so that the tightest ones are studied first. The order of constraints in \( CL \) will be \([C_5, C_2, C_4]\) and the first checked constraint is \( C_5 \). This constraint is violated so it generates a nogood, \( X_3 \) is selected to change its current value, the constraints related with \( X_3 \), \([C_2, C_3]\), are added to \( CL \) and \( CL \) is reordered. The current \( CL \) to check is: \([C_5, C_2, C_4, C_3]\). The constraint \( C_5 \) is checked again. \( C_5 \) is still violated and \( X_3 \) chooses another value from its current domain. After \( X_3 \) chooses its value, the constraint \( C_5 \) is checked again and this time, \( C_5 \) is consistent and it is removed from \( CL \). The next selected constraint is \( C_2 \) which is not violated, so it is removed from \( CL \) and we check the next constraint \( C_4 \). This constraint is violated, \( X_4 \) is selected to change its current value and \([C_3]\) is added to \( CL \) and reordered. The current \( CL \) to check is: \([C_4, C_3]\). Remaining constraints in \( CL \) are satisfied, then the solution is found.

We observe that standard \( PDB \) checked 8 constraints, but \( PDB \) using \( DCO \) checked just 7 constraints. It is true that the difference is not great because the problem is small, but for large problems, the difference will be clearer and we will see this in the next section.
6 Experimental results

Meetings are an essential part of running a successful business [2]. The process of scheduling these meetings can be quite complicated as organizing a meeting with many participants can lead to many conflicts. In simple terms, it involves searching for a time and place when and where all the meeting participants are free and available.

Formally, a MSP is defined by the following parameters:
• \( P = \{ p_1, p_2, \ldots \} \), the set of participant.
• \( S = \{ s_1, s_2, \ldots \} \), the set of calendar for each participant.
• \( M = \{ m_1, m_2, \ldots \} \), the set of meetings.
• \( \text{At} = \{ at_1, at_2, \ldots \} \), the set of collections of participants that define which attendees must participate in each meeting, i.e. people in \( at_i \) must participate in the meeting \( m_i \), \(1 < i < k\) and \( at_i \in P\).
• \( L = \{ l_1, l_2, \ldots \} \), the set of location where meetings can be scheduled.

The meeting scheduling problem as described above can be naturally modelled as a Constraint Satisfaction Problem (CSP) in the following way:
• Set of n variables \( X \): \{ meetings to be scheduled for each participant \( x_{mp} \) with \( m \in M \) and \( p \in P \) (i.e. number of variables is \( P \times M \) ) \}.
• Set of Domains \( D \): \{ weekly time slots for each participant \}.
• Set of constraints \( C \): \{ 1) For every pair of variables \( (x_{mp}, x_{mq}) \) with \( p \neq q \in P \) there is an equality constraint \( x_{mp} = x_{mq} \), if there is two participants \( p \) and \( q \) participate in same meeting \( m \). 2) For every pair of variables \( (x_{mp}, x_{np}) \) with \( m \neq n \in M \) there is an Arrival Time constraint \( t_{nm} (| x_{mp} - x_{np} | \geq t_{nm}) \), if there is a participant \( p \) participates in both meetings \( m \) and \( n \) \}.

Simplifying assumptions:
• All participants have the same size of weekly calendar.
• All Meetings have the same duration (1 time slot).
• Each participant attends the same number of meetings.

This problem can be defined as a sequence of static MSPs: \( MSP_0, MSP_1, ..., MSP_{\alpha-1}, MSP_\alpha \), where the network is subject to constraints perturbations, and each \( MSP_i \) differs from the previous one by the modification of a set of constraints.

The dynamic MSP (\( DMSP \)) can introduce two different types of changes:
• Additional Arrival Time constraint (traffic jams, train rescheduling, etc.)
• New links between meetings previously unconnected (a participant of meeting \( m_i \) is required to attend another meeting \( m_j \) as well).

Our experimental evaluation of \( DMSPs \) introduce the first type of change.
We have carried out a set of experimental tests on random instances of the Meeting Scheduling in order to show how DCO may affect the performance of the PDB algorithm. We evaluate the results in terms of number of constraint checks (Ccs), computation time (Time) and solution stability (Stability).

We evaluate the performance of the $PDB_{DCO}$ algorithm with the standard PDB on DMSP problem. This problem is characterized by $< m, n, p, d, h, t, a >$, where $m$ is the number of meetings, $n$ is the number of meetings per participant, $p$ is the number of participants and $d$ determines the number of days. Different time slots are available for each meeting, and $h$ is the number of hours per day, $t$ is a duration of the meeting and $a$ is the percentage of availability for each participant.

We present our results for the class $< 20, 5, 15, 5, 10, 1, 90 >$ and we vary the rate of injected constraints (%IC) (i.e. Arrival Time Constraint) from $\sim2\%$ to $\sim24\%$. We generated 25 different instances solved by each algorithm and the results are an average of these 25 runs. All experiments were performed on the Java platform.

Figure 5: Number of Constraint Checks performed by standard PDB and PDB using DCO

Figure 5 presents the computational effort in terms of constraint checks needed to repair solutions. Since the selection of the constraint to check in $PDB$ is done randomly, this last appears inconstant compared with $PDB_{DO}C$. 
Generally, the most relevant heuristic for dynamically changed environments is $C_{weight}$, because it tries to satisfy first the constraint that led to a failure most of the time.

The second heuristic that presents interesting results is $C_{ddeg}$. Since the heuristic $C_{ddeg}$ selects the constraint with the variable that has more constraints with unassigned variables, we can say that this heuristic has been able to focus research on the hardest branches of the network to correct solution as soon as possible.

However the heuristic $C_{deg}$ selects the constraint with the variable that has more constraints with all variables (be affected or not) and, as we will see in the figure of solution stability, this heuristic disturbs the affectations of assigned variables and makes the process of correction unstable (the number of $Ccs$ sometimes increase and sometimes decrease).

As knowing, more the tightness is large more the number of allowed tuples is small. The heuristic $C_{tight}$ classifies the constraints so that the tightest constraints are classified first. Thus, a $PDB_{DCO}$ algorithm tackles the hardest part of a search problem first and inconsistencies are found earlier and the number of constraint checks is significantly reduced. We can say that this heuristic, like $C_{dom}$, tries to find a new instantiation, the one likely to fail as quickly as possible.

![Figure 6: Time performed by standard PDB and PDB using DCO](image)
Dynamic constraint ordering heuristics for dynamic CSPs

Figure 6 shows the computational time performed by each heuristic for the same problems. The really interesting result is that PDB with the Cweight, Ctight and Cdom heuristics need less time in order to repair solutions. Note that Cddeg needs a lot of time to correct the solution and it is because of computing a dynamic degree of each constraint at each moment. Concerning Cdeg and the standard PDB, these heuristics spend more time to revise the solution because of their instability.

![Figure 7: stability performed by standard PDB and PDB using DCO](image)

Figure 7 shows the stability of solution performed by each heuristic. As we said, the stability of solution measure the number of variables which get different values in both solutions. We evaluated the performance of the proposed heuristics using the DMSP problems with $n = 5$ is the number of meetings per participant, $p = 15$ is the number of participants which mean that the number of variables is $\text{card}(X) = n \times p = 75$ variables.

The experiments show that PDB with Cdeg and standard PDB disturb the affectation of an important number of variables compared to Cweight and Ctight that maintain the stability of solution. For example, when the rate of injected constraints is equal to 16% (~24 injected constraints), Cdeg changes the values of 31 variables (~41% of the solution), however Cweight changes only the values of 21 variables (~28% of the solution).
7 Conclusion

In this paper, we have presented a dynamic constraint-based heuristic for repairing solution in dynamic CSPs, called Dynamic Constraint Ordering DCO. Our approach is able to handle the case where a DCSP problem may have changed constraints, regarding the dynamic elements of the environment, and old solution needs to be quickly repaired when new information is received. The proposed heuristic dynamically reorders the constraints that have to be firstly satisfied. This technique is able to guide the repairing algorithm, namely Partial-Order Dynamic Backtracking PDB, towards a most relevant constraints network subproblem. The experiments that we have performed, prove that our repair-directed approach is interesting, and the results show clearly the efficiency of these heuristics combined with a revised version of PDB algorithm in MSP instances in terms of running time, computation effort and solution stability. As Future work, we are looking at how some of Dynamic Constraint Ordering heuristics can be applied to other domains. As such, we plan to examine the efficiency of these techniques in the framework of Minimal Perturbation Problems and we believe that this union can lead to very effective solutions to a wide range of complex dynamic problems.

References


Received: November 1, 2013