Dynamic Global Constraints in Backtracking Based Environments

Roman Barták*

Charles University, Faculty of Mathematics and Physics
Malostranské námì stí 2/25, 118 00 Praha, Czech Republic

phone: (+420-2) 2191 4242
fax: (+420-2) 2191 4323
e-mail: bartak@kti.mff.cuni.cz

* Supported by the Grant Agency of the Czech Republic under the contracts no. 201/99/D057 and 201/01/0942. I would like to thank Petr Vilím for implementing the `alldifferent` constraint and for performing the empirical evaluation. I am also grateful to Ondøej Èepek for proof reading of the paper draft.

Abstract

Global constraints provide strong filtering algorithms to reduce the search space when solving large combinatorial problems. In this paper we propose to make the global constraints dynamic, i.e., to allow extending the set of constrained variables on flow. We describe a generic dynamisation technique for an arbitrary monotonic global constraint and we compare it with the semantic-based dynamisation for the `alldifferent` constraint. At the end we sketch a dynamisation technique for non-monotonic global constraints. A comparison with existing methods to model dynamic problems is given as well.

Keywords: global constraints, filtering algorithm, dynamic models

Constraint programming (CP) is successfully applied to real-life combinatorial (optimisation) problems thanks to its declarative character, which supports natural modelling of real-life problems, and thanks to the general solving technology, which can encapsulate various solving techniques. One of the main features of CP is locality of the problem description i.e. we model the problem using a set of constraints binding small sets of variables which are much smaller than the set of all variables in the problem. This is different from the traditional operation research (OR) approach where the “constraints” bind all the variables and thus, OR solving methods can exploit global reasoning about the problem. On the other hand, global reasoning can be very expensive and a simple method of constraint propagation based on domain filtering can reduce the search space more efficiently.

The goal of constraint propagation is to achieve some level of consistency in the network of constraints and variables by removing inconsistent values from variables' domains (an inconsistent value cannot take part in any solution). Achieving higher level of consistency leads to removing more inconsistent values but it is more expensive in terms of time and space. Consequently, a simple arc consistency or its generalised version for n-ary constraints is usually used. Application of CP to many real-life problems shows that a good trade-off between efficiency and level of consistency can be achieved by using global constraints instead of sets of binary constraints, e.g. Simonis (1999).

A global constraint encapsulates several simple constraints and by exploiting semantic information about this set of constraints it can achieve stronger pruning of domains. Filtering algorithms for global constraints are based on methods of graph theory, discrete mathematics, or operation research so they make the bridge between these mathematical areas and search-based constraint programming with origins in artificial intelligence. However, the traditional global constraints require all the constrained variables to be known before the constraint is posted to the system. This complicates usage of global constraints in areas where new variables are generated during the course of solving. This difficulty can be solved using dummy variables that represent slots for variables to come. Unfortunately, this approach cannot be used if large number of dummy variables is required because it decreases efficiency of the filtering algorithm and it increases memory consumption.

In Barták (2001) we proposed a dynamic view of global constraints that is worked out in more details here. Such a dynamic global constraint allows adding a new variable(s) during the course of problem solving and removing this variable(s) upon backtracking. Thus, a dynamic global constraint can be posted before all the constrained variables are known which brings the advantage of earlier domain pruning. As far as we know this is the first attempt to formalise such a dynamic behaviour of global constraints. We designed a straightforward generic technique for making monotonic global constraints dynamic. This technique is easy to implement in the current constraint systems, as it does not require a change of paradigm from CSP to Dynamic CSP and it can exploit directly the existing filtering algorithms for global constraints. However, the generic dynamisation suffers from time and space inefficiencies so we propose to include dynamic behaviour directly within the filtering algorithm.

Our dynamisation techniques are appropriate for monotonic global constraints in backtracking based environment. It means that adding new variables to the constraint does not enlarge the solution set of this constraint (monotony) and the variables can be removed only upon backtracking. This simplifies the implementation while still providing the advantage of earlier domain pruning. Our approach is thus useful for constraint logic programming (CLP) systems where search naturally interleaves with problem formulation (i.e.,

with defining the variables and posting the constraints). Nevertheless, it can be applied to arbitrary constraint solving system provided that this system follows the ideas behind CLP.

The paper is organised as follows. In Section 1 we give a motivation for our research and we compare our approach with existing techniques to solve dynamic problems. In Section 2 we formally introduce the basic notions and terminology used within the paper. Section 3 is dedicated to the description of a generic technique for making the global constraints dynamic. We prove the soundness and completeness theorem there that can serve as a theoretical foundation for semantic-based dynamisation as well. In Section 4, we present an example of semantic-based dynamisation, in particular we extend the well-know filtering algorithm for the `alldifferent` constraint to behave dynamically. We theoretically compare time and space complexity of generic and semantic-based dynamisations of the `alldifferent` constraint and we also present an empirical comparison of our approach with the technique based on dummy variables. In Section 5 we describe a basic idea how non-monotonic constraints can be made dynamic using an application of our approach to a monotonic approximation of the constraint. We conclude with some references to existing implementations of constraint satisfaction from the point of view of modelling dynamic problems.

1 Motivation and related works

A traditional formulation of a constraint satisfaction problem is static in sense that all the variables and the constraints are known before we start to solve the problem. Global constraints follow this static definition so the global constraint can be posted when all the constrained variables are known. Nevertheless, there exist problems that do not fit this static model and that require a more dynamic formulation of the problem. We can roughly classify these problems into two categories.

In the first category, the problem is dynamically modified from external environment, e.g., an existing solution should be adapted to include new information (like broken machine during scheduling), or interaction with the user is required during problem solving. Such problems are called reactive, on-line, or incremental constraint satisfaction, for examples see Fages, Fowler, and Sola (1995) and Van Hentenryck (1990). On-line constraint satisfaction is not the problem area that we deal with primarily in this paper, even if our ideas can be applied to the on-line problems provided that the variables are added and removed in LIFO (last in first out) manner.

The second category covers problems that evolve dynamically during the solving process. For example in configuration or planning problems, the variables and constraints might change as the search progresses. We have identified this type of dynamic problems in scheduling of complex process industries where so-called process-dependent activities appear, see Barták (2000). Existence of the process-dependent activity depends on allocation of other activities. In terminology of CSP, existence of some variables and constraints (that describe the activity) depends on values of other variables (on allocation of other activities). Pegman (1998) described a similar type of problems in steel-making industry. Figure 1 gives two examples of process-dependent activities.

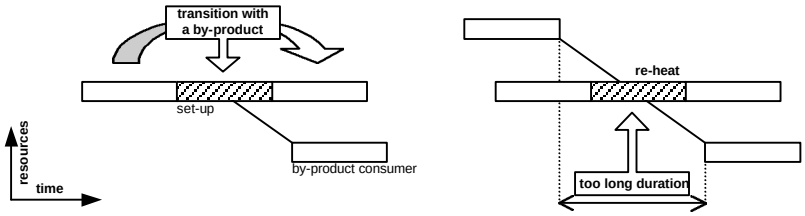

Figure 1. Gantt charts with set-up (left) and re-heating (right) activities. In both cases a special activity (striped rectangle) is necessary because this activity consumes resource capacity or it is connected to activities in other resources.

Solving such dynamic constraint satisfaction problems is a hot topic of research in CP, however, not a new one. Mittal and Falkenhainer (1990) proposed a concept of Dynamic Constraint Satisfaction motivated by problems in configuration. Since then the idea of using activity constraints and dummy variables is used in various concepts. This approach is useful when the number of dummy variables is not very large, i.e., when the number of alternatives is linear rather than exponential. Figure 2 shows a structure of such problem formulation in CLP. Nodes represent the choice points, that cause adding new variables and/or constraints to the problem, while edges describe introduction of these variables and constraints.

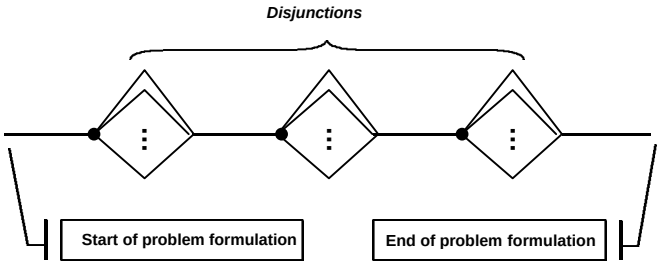

Figure 2. A typical problem formulation in CLP with disjunction represented using alternative clauses (the problem is formulated from left to right with nodes describing the choice points).

During the problem formulation (left to right in Figure 2) one of the alternative paths is selected. Another alternative is selected when labelling fails. Hentenryck and Deville (1991) identified that such interleaving of problem formulation with labelling is not a good way of defining disjunctive problems in CLP due to weak constraint propagation. Therefore they proposed a cardinality operator that can handle the disjunction more actively. Similar idea can be find in Van Der Linden (2000), i.e., all the constraints and all the variables are posted first and some of them (representing the alternatives that has not been not selected) are deactivated during labelling. Nevertheless, this approach using dummy variables (and disjunctive constraints) can hardly be applied to problems with highly dynamic structure like planning. In these problems the number of alternatives is very large because it is impossible to predict which variables and which constraints will be used in which combinations, see Nareyek (2000). Figure 3 shows a structure of such problem formulation in CLP using the same notation as Figure 2. We call these problems highly dynamic.

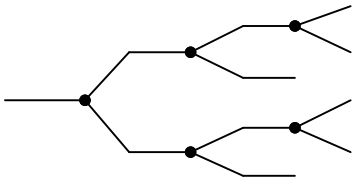

Figure 3. Formulation of highly dynamic problems forms a tree with many alternatives. Every branch of this tree represents a constraint satisfaction problem with different set of variables and constraints.

Nareyek (1999) proposed a concept of Structural Constraint Satisfaction to solve the highly dynamic problems. The difficulty of this concept is that the constraint satisfaction package must be programmed from scratch to support it. Therefore it is almost impossible to modify an existing system to solve the problems using the Structural Constraint Satisfaction framework.

Another possibility to solve the highly dynamic problems is to interleave the problem formulation with labelling. This can be done easily in the widespread CLP framework. However, this technique suffers from efficiency problems as we described above. One of these efficiency problems here is that usage of powerful global constraints is limited to situations when the constrained variables are known. In particular, we have to wait until all the constrained variables are known before we can post the global constraints. In this paper we concentrate on solving this difficulty by allowing a dynamic formulation of the global constraint, i.e., it is possible to add new variables to the constraint as the search progresses and to remove them upon backtracking. Thus the global constraint can be posted and propagated earlier, which improves domain filtering. Note that this concept of dynamic global constraints can be naturally implemented in existing CLP environments.

Dynamic Constraint Satisfaction by Mittal and Falkenhainer (1990) also deals with adding and removing variables and constraints to/from the problem. Addition and retraction can be done in arbitrary order and, thus, the concept must be implemented from scratch and it is hard (or impossible) to extend existing CSP packages to support it. In our dynamic global constraints we do not need generality of Dynamic CSP (variables are added/removed to/from the constraint in LIFO manner) that simplifies the implementation and integration to existing systems.

2 Preliminaries

A *Constraint Satisfaction Problem* P=(X,D,C) is defined as a set of variables $X=\{x_1,\ldots,x_n\}$, a set of domains $D=\{D(x_1),\ldots,D(x_n)\}$, where $D(x_i)$ is a finite set of possible values for variable x_i, and a set of constraints $C=\{c_1,\ldots,c_m\}$. A *constraint* c_i on the ordered set of variables $X(c_i)= (x_{i1},\ldots,x_{ik})$ is a subset of the Cartesian product $D(x_{i1})\times\ldots\times D(x_{ik})$, it specifies the allowed combinations of values for the variables $x_{i1},\ldots,x_{ik}$. $|X(c_i)|$ is called arity of the constraint c_i.

Assume that $(v_1,\ldots,v_n)\downarrow(x_{i1},\ldots,x_{ik})$ is a standard projection operator, i.e., a projection of the tuple $(v_1,\ldots,v_n)$ to the variables $(x_{i1},\ldots,x_{ik})$: $(v_1,\ldots,v_n)\downarrow(x_{i1},\ldots,x_{ik}) = (v_{i1},\ldots,v_{ik})$. We call a complete instantiation $(v_1,\ldots,v_n)$ of the variables, such that $\forall i=1,\ldots,n\ v_i\in D(x_i)$ and $\forall j=1,\ldots,m\ (v_1,\ldots,v_n)\downarrow X(c_j) \in c_j$, a *valid tuple* or *solution*, i.e., the valid tuple is a complete instantiation of the variables satisfying all the constraints. Sol(P) is a set of all valid tuples, we call it a *solution set* of the constraint satisfaction problem P.

Problem P'=(X',D',C') is called an *extension of the problem* P=(X,D,C), if $X\subseteq X'$, $C\subseteq C'$ and $\forall x\in X$ $D(x)=D'(x)$. Briefly speaking, we extend the problem if we add new variables and constraints to it. It is known

that constraint satisfaction is monotonic, i.e., every solution of the extended problem is also a solution of the original problem. Formally, if P' is an extension of P, then $Sol(P')\downarrow X \subseteq Sol(P)$, where X is a set of variables in P.

A *global constraint scheme* G is a function that maps arbitrary finite ordered set of variables to a conjunction of constraints. We call such a conjunction of constraints a global constraint, i.e., the global constraint forms a sub-problem in the constraint satisfaction problem. Note, that the constraints in $G(x_1,\ldots,x_k)$ may contain variables other than $x_1,\ldots,x_k$ but these variables are local/hidden there, i.e., they do not appear in the rest of the problem. If we have a global constraint G(X), we say that G(Y) is its extension if $X \subseteq Y$. Let Sol(G(X),D(X)) be a solution set for the global constraint G(X) and a set of variables X with domains D(X). A global constraint scheme is *monotonic*, if $Sol(G(X\cup Y),D(X\cup Y))\downarrow X \subseteq Sol(G(X),D(X))$ for arbitrary sets X and Y of variables. Monotonicity means that extending a global constraint, i.e., adding new variables to the global constraint, does not add new solutions[1].

Example 1: The *alldifferent* global constraint scheme is defined in the following way

$$\mathbf{alldifferent}(x_1,\ldots,x_n) = \bigwedge_{\substack{i=1..n \\ j=1...n \\ i\neq j}} x_i \neq x_j$$

There are no local variables in the `alldifferent` constraints and this global constraint scheme is monotonic.

Example 2: The *atleast* global constraint scheme is defined in the following way

$$\mathbf{atleast}(k,y,\{x_1,\ldots,x_n\}) = \bigwedge_{i=1,\ldots,n}(b_i \text{ in } \{0,1\}) \wedge \left(\bigwedge_{i=1,\ldots,n} x_i = y \Leftrightarrow b_i = 1\right) \wedge \sum_{i=1}^{n} b_i \geq k$$

In this scheme, there are *n* local variables b_i with the domain {0,1}. This global constraint scheme is not monotonic (e.g., *atleast*(2,1,{1,0,1}) is satisfied, but *atleast*(2,1,{1,0}) is not satisfied).

According to Rossi, Dahr, and Petrie (1990), an arbitrary *n*-ary constraint including the global constraints can be decomposed into an equivalent set of binary constraints for which many consistency techniques were designed. Nevertheless, these consistency techniques, like arc consistency, can usually be extended to *n*-ary constraints, see Bessiere and Régin (1999). As noted by Bessiere (1999), this could be expensive in general but for global constraints there exist efficient filtering algorithms exploiting semantic of the constraint, for examples see Régin (1994), Régin (1996) and Simonis (1999). Therefore global constraints are very important for solving real-life problems.

We say that the global constraint G(X) with domains D(X) is *A-consistent* iff domains D(X) together with the constraint G(X) satisfy an A-consistency condition. For example, $G(x_1,\ldots,x_n)$ is arc consistent iff $\forall i=1,\ldots,n\ \forall v_i \in D(x_i)\ \forall j\neq i\ \exists v_j \in D(x_j)$ s.t. $(v_1,\ldots,v_n)\in G(x_1,\ldots,x_n)$. *A-filtering algorithm* for a global constraint G is a mapping *cons* of the global constraint G(X) and domains $D(x_i)$ of its variables to the domains $D'(x_i)\subseteq D(x_i)$ such that G(X) with D'(X) is A-consistent,. Usually, we require the filtering algorithm to be sound, i.e., $Sol(G(X),D(X))\subseteq cons(G(X),D(X))$, and monotonic, $\forall x\in X\ D'(x)\subseteq D(x) \Rightarrow cons(G(X),D'(X))\subseteq cons(G(X),D(X))$.

[1] It seems that monotonicity feature of the global constraints is closely related to decomposable constraints studied in Gent, Stergiu, and Walsh (2000).

Soundness means that no solution is removed by filtering. Monotony means that if filtering starts with larger domains then it does not achieve better pruning of the domains than when starting from smaller domains.

Our objective is to design a framework that allows extending the set of variables in the global constraint scheme on fly while preserving consistency achieved by the filtering algorithm for the static global constraint. In particular, we design a dynamic extension of the *A*-filtering algorithm which maps a global constraint G(X), domains D(X) and a new set of variables Y to domains D'(X∪Y) such that Sol(G(X∪Y),D(X∪Y))⊆D'(X∪Y)⊆*cons*(G(X∪Y),D(X∪Y)), where *cons* is a static *A*-filtering algorithm for G.

3 A generic dynamisation technique

Global constraints are used in constraint programming systems in the same way as other constraints are used, but global constraints can be defined over arbitrary finite set of variables. It means that if we know the constrained variables, we can post a global constraint among them. Nevertheless, in some problems we do not know all the variables in advance and new variables are introduced as the search progresses. Consequently, we cannot post a global constraint until we know all the variables. Nevertheless, if the global constraint is monotonic we can do better by posting a global constraint over the known variables and when a new variable arrives we can post a new global constraint extended by the new variable and deactivate the old constraint. This is the basic idea behind a generic dynamisation technique that we will describe in details now.

Assume that there exists a filtering algorithm for the global constraint *G* that is wrapped in a trigger procedure *g*. In particular, procedure $g(x_1,\ldots,x_n)$ calls the filtering algorithm for the constraint *G* every time a domain of any variable x_i is changed[2]. We will use this procedure as the basic interface to the filtering algorithm for *G*. Note also that we propose an algorithm for extending the set of variables. Variables can be removed from the constraint only upon backtracking, i.e. LIFO (last-in first-out) mechanism is applied for adding/removing variables to/from the constraint. Moreover, if we add a set of variables, the whole set is removed together upon backtracking. LIFO mechanism for adding/removing the variables may seem like a restriction but it fits perfectly in the CLP framework and it simplifies significantly the implementation.

Our algorithm exploits the monotony property of the global constraint, in particular we can start with the previous solution (domains) to get a solution (restricted domains) of the extended constraint. The algorithm first deactivates the current propagation algorithm (step 1), note that it does not mean removing the constraint from the constraint store, it is just stopping propagation through it (the internal data structures are kept). Then, it saves domains of the variables to the stack (step 2) - typically, the underlying constraint engine saves such information; we highlight it just to illustrate the track of the algorithm. Finally, it posts a new global constraint with the extended set of variables (step 3).

[2] This is a standard way of connecting a new consistency algorithm to the constraint engine in CLP, see Carlsson, Ottoson, and Carlsson (1997). It is up to the underlying constraint engine to plan calls of the filtering procedures but such techniques are out of scope of this paper, so we do not go in detail there.

Algorithm 1: ADDVARIABLE($G(X_1,\ldots, X_k)$, X_{k+1})

1) deactivate $g(X_1,\ldots, X_k)$,
   i.e., push its internal data structures to the stack and stop propagation through this constraint (if $k\geq 1$, otherwise do nothing)
2) push domains of $X_1,\ldots, X_k$, X_{k+1} to the stack
3) post $g(X_1,\ldots, X_k, X_{k+1})$ to the constraint store

When removing a variable upon backtracking, we simply reverse Algorithm 1. It means: removing the extended constraint from the constraint store (now, it is real removing, not just deactivation), restoring domains of the variables (this is usually done automatically when the extended constraint is removed from the constraint store), and activating the former constraint again. Note that in CLP no special algorithm is necessary to implement variable removal, standard backtracking technique does this job.

The proposed dynamisation technique is very general and it is applicable to arbitrary monotonic global constraints. In fact, we can use an off-shelf code and wrap it using meta-programming techniques to get a dynamic version of the global constraint. The following theorem ensures that this technique is sound (no solution is lost) and good enough in comparison with static filtering. The advantage over the static filtering is that the user may post the global constraint earlier before the complete set of variables is known.

Theorem: If the static filtering algorithm *cons* is sound and monotonic and the global constraint G is monotonic then the algorithm ADDVARIABLE is sound, i.e., it does not remove any valid tuple, and it achieves at least the same pruning as static filtering. Formally:

$Sol(G(X\cup Y),D(X\cup Y))\subseteq$ *cons*$(G(X\cup Y),D'(X)\times D(Y))\subseteq$ *cons*$(G(X\cup Y),D(X\cup Y))$, where

$D'(X)=$ *cons*$(G(X),D(X))$.

Proof:

1.	$D'(X)\subseteq D(X)$	a feature of *cons*
2.	***cons*(G(X∪Y),D'(X)×D(Y)) ⊆ *cons*(G(X∪Y),D(X∪Y))**	monotony of cons + $D'(X)\times D(Y) \subseteq D(X)\times D(Y) = D(X\cup Y)$
3.	$Sol(G(X),D(X))\subseteq D'(X)$	soundness of *cons*
4.	let $s \in Sol(G(X\cup Y),D(X\cup Y))$	
5.	$s{\downarrow}X \in Sol(G(X),D(X))$	monotony of G
6.	$s{\downarrow}X \in D'(X)$	3+5
7.	$s \in D'(X) \times D(Y)$	
8.	$s \in Sol(G(X\cup Y), D'(X)\times D(Y))$	4+7+feature of Sol
9.	$Sol(G(X\cup Y),D(X\cup Y)) \subseteq Sol(G(X\cup Y), D'(X)\times D(Y))$	4-8
10.	$Sol(G(X\cup Y), D'(X)\times D(Y))\subseteq$ *cons*$(G(X\cup Y),D'(X)\times D(Y))$	soundness of cons
11.	**$Sol(G(X\cup Y),D(X\cup Y)) \subseteq$ *cons*$(G(X\cup Y),D'(X)\times D(Y))$**	9+10

❑

4 A semantic-based dynamisation

The dynamisation technique proposed in the previous section suffers from some inefficiency caused by posting a new constraint after extending the set of variables:

1) the new constraint must build its internal data structures from scratch, which increases the time complexity,
2) we keep the data structures of the old constraint before extension as well as the data structures of the new constraint after extension, which increases the space complexity,
3) if the existing interface to the filtering algorithm does not support deactivation (and this is the common case) then propagation is duplicated, i.e., a filtering algorithm is called for all instances of the global constraint even if calling it for the instance with the largest set of variables is enough.

To eliminate the above mentioned difficulties we propose including dynamic extendibility into the global constraint itself. We believe that this could be done at least for some global constraints as we show in the next section for a well known `alldifferent` constraint.

4.1 A dynamic `alldifferent` constraint

Régin (1994) proposed an efficient filtering algorithm for the `alldifferent` constraint. His implementation is based on matching theory, in particular on matching over bipartite graphs. The bipartite graph for the `alldifferent` constraint is called a value graph and it is defined in the following way:

Definition 1: Given an `alldifferent` constraint C, the bipartite graph $GV(C) = \langle X_C, D(X_C), E\rangle$, where X_C is a set of variables in C, $D(X_C)$ is a union of domains D_i for all variables in X_C and $(X_i,a)\in E$ iff $a\in D_i$, is called a **value graph** of C.

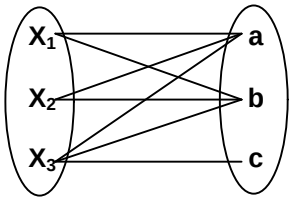

Figure 4. A value graph for the `alldifferent` constraint with three variables.

The filtering algorithm for the `alldifferent` constraint computes the maximal matching and removes edges that are not part of any maximal matching. Note that removing an edge from the value graph is equivalent to removing a value from the variable domain. Moreover, if the maximal matching does not cover all the variables from the `alldifferent` constraint then the constraint fails (it is not possible to find a consistent labelling of variables).

Algorithm 2: AllDiff-Initialisation(C)

```
1) build G = ⟨X_C, D(X_C), E⟩,
2) M(G) ← ComputeMaximumMatching(G)
3) if |M(G)|<| X_C | then return false
4) RemoveEdgesFromG(G,M(G))
5) return true
```

The procedure RemoveEdgesFromG deletes every edge that does not belong to any matching which covers X_C. Such edges are found by exploring even alternating paths and cycles and by looking for strongly connected components in the value graph. The description of the algorithm linear in the number of edges can be found in Régin (1994).

Naturally, the constraint systems consist of more than a single `alldifferent` constraint and the other constraints may reduce the domains of variables from X_C as well. We can repeat the AllDiff-Initialisation procedure each time the domain of any variable from X_C is changed (a value is deleted) but we can do better by using the fact that before the deletion, a matching which covers X_C is known. The algorithm by Régin (1994) uses a function MatchingCoveringX(G, M_1, M_2), which computes a matching M_2 covering X_C from a matching M_1, which is not maximal. If no such matching exists then the procedure returns *false*. We present here a slightly simplified version of the algorithm from Régin (1994) (we expect that when an edge (X,a) is removed from the graph then the value a is not the only value for the variable X, otherwise the constraint that caused such reduction has already failed). The algorithm gets the value graph G, the original maximum matching $M(G)$, and the list of edges ER to delete as input. It returns *false* if there is no maximum matching which covers X_C, and it returns *true* otherwise and deletes every edge that does not belong to any matching which covers X_C.

Algorithm 3: AllDiff-Propagation(G,M(G),ER)

```
1) computeMatching ← false
2) for each e∈ER do
3)      if e∈M(G) then
4)        M(G) ← M(G) - {e}
5)        computeMatching ← true
6)      remove e from G
7) if computeMatching then
8)      if ¬MatchingCoveringX(G,M(G),M') then
9)        return false
10)       else
11)         M(G) ← M'
12)     RemoveEdgesFromG(G,M(G))
13)     return true
```

Algorithms 2 and 3 can be used for initialisation and propagation of dynamic version of the `alldifferent` as well, provided that we can extend the value graph after adding new variables. Remember that the `alldifferent` constraint is monotonic so we can add a new variable (or a set of variables) to the constraint without extending the solution set. Moreover, we can incrementally extend the value graph to get a value graph for the constraint with more variables. In fact we are extending the set X_C of variables, the set $D(X_C)$ of values (perhaps), and the set E of edges. The new edges connect only the new variables with values (old or new); there is no new edge connecting any old variable with any value (we are not extending the domains of the old variables so we keep monotonicity).

Definition 2: Let $\langle X, D(X), E\rangle$ be a value graph of the `alldifferent` constraint $C(X)$ for the set X of variables. Then the bipartite graph $\langle X\cup Y, D(X\cup Y), E'\rangle$ where Y is set of added variables and $E' = E \cup \{(X_i,a) \mid X_i \in Y \,\&\, a \in D_i, \}$ is called an **extended value graph** of C($X\cup Y$) for the extended set $X\cup Y$ of variables.

An important thing is that the extension of the variable set does not influence the decisions taken before. In particular, if any edge is deleted from the original value graph (because it does not belong to any matching which covers X) then this edge does not belong to any matching which covers $X\cup Y$ (the constraint is monotonic). This observation is used in the algorithm that updates the value graph after adding new variables. It means that we can incrementally update the value graph after adding new variables instead of computing a new value graph from scratch.

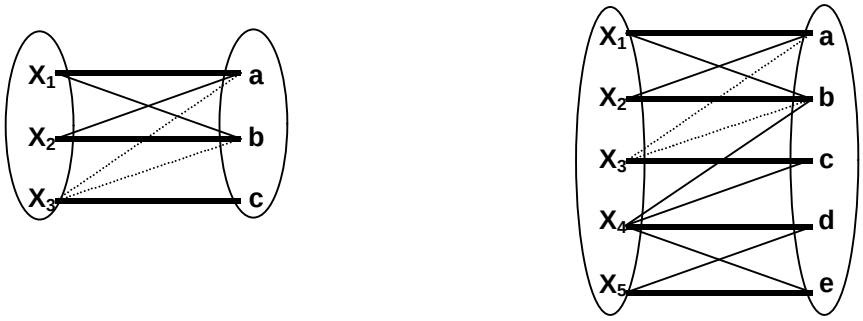

Figure 5. Edges deleted (dotted edges) from the original value graph (left) do not belong to the extended value graph (right). Bold edges are the maximum matching edges.

After extending the original value graph by adding new edges, the former maximum matching does not cover the extended set $X\cup Y$ of variables. We can use the function MATCHINGCOVERINGX to extend the known matching that covers X to a matching that covers $X\cup Y$. If there exists a maximum matching that covers $X\cup Y$ then the edges that do not belong to any maximum matching are removed using REMOVEEDGESFROMG and the algorithm returns *true*, otherwise it returns *false*.

Algorithm 4: ALLDIFF-UPDATE(G,M(G),EA)

```
1) for each e∈EA do
2)     add e to G
3) if ¬MATCHINGCOVERINGX(G,M(G),M') then
4)     return false
5) else
6)   M(G) ← M'
7) REMOVEEDGESFROMG(G,M(G))
8) return true
```

Removing a variable from the `alldifferent` constraint is done upon backtracking only so we do not present a special algorithm for this. The standard mechanism with a stack to recover data structures can be used there. To prove soundness of the ALLDIFF-UPDATE algorithm we can use the results from Régin (1994) combined with the monotonicity feature of the `alldifferent` constraint.

4.2 Complexity analysis

Besides the dynamic filtering for the `alldifferent` constraint (presented in Section 4.1) we can use a generic dynamisation technique proposed in Section 3. Let us now compare the complexity of these two methods for making the `alldifferent` constraint dynamic. We will use the basic complexity results from Régin (1994) rather than describing all the details about the data structures etc. This is enough for our purpose that is to compare two approaches sharing the same "low-level" procedures.

Assume that p is a number of variables in the constraint and d is a number of all different values in the domains of these variables (a size of the union of the domains). Thus $p+d$ is a number of vertices in the value graph. Let m be a number of edges in the value graph, clearly $m \leq pd$ (the value graph is a bipartite graph) so we approximate m by pd.

Space complexity. To represent the `alldifferent` constraint we need to keep its value graph and the maximum matching. The graph can be represented by the set of its edges so the space complexity is $\boldsymbol{O(m)}$ i.e. in the worst case $\boldsymbol{O(dp)}$. The maximum matching consumes $\boldsymbol{O(p)}$ space, e.g., the matching can be represented by the list of values in the matching. If a generic dynamisation technique is used then a new value graph is introduced (and the previous value graph is kept in memory) after adding a new variable. Thus, the worst space complexity of adding a new variable is $\boldsymbol{O(dp)}$.

In case of a dynamic filtering algorithm, only new edges are added to the value graph when the new variable arrives. These edges can only connect the new variable with the values, so the maximal number of the new edges is d. Remember that we also need to keep the maximum matching. Thus, space complexity is $\boldsymbol{O(d+p)}$ that is much better than generic dynamisation.

Time complexity. According to Régin (1994), time complexity of ALLDIFF-INITIALISATION is $O(dp\sqrt{p})$. Thus, time complexity of adding a new variable using the generic dynamisation technique is $O(dp\sqrt{p})$ as well (in fact, we are adding a completely new constraint while keeping the former constraint).

As shown in Régin (1994), time complexity of REMOVEEDGESFROMG is $O(m+d+p)$ and time complexity of MATCHINGCOVERINGX is $O(m\sqrt{k})$ where k is a number of edges missing in the maximal covering (these edges must be added to find a covering that covers all the variables). Note that if we are adding a new variable to the constraint then exactly one edge is missing in the maximal covering (i.e., k=1) - the edge connecting the new variable with some value. Consequently, time complexity of MATCHINGCOVERINGX called in ALLDIFF-UPDATE is $O(m)$. Together, time complexity of adding a new variable using ALLDIFF-UPDATE is $O(m+d+p)$. In the worst case this complexity is $O(dp)$, still $\sqrt{p}$ -times better than generic dynamisation.

Note that if we know the variables in advance then it is more efficient to add them all together, time complexity is $O(dp\sqrt{p})$, than adding them incrementally (one by one), time complexity is $O(dp^2)$. Algorithm 4 supports adding more variables together in an efficient way with time complexity $O(dp\sqrt{k})$, where k is a number of added variables.

Time complexity of propagation of single deletion (a value is removed from the domain of a variable by another constraint) through the constraint is $O(dp)$, Régin (1994), same for both methods of dynamisation. Nevertheless, note that some systems do not support deactivating the constraint from the constraint store - the constraint is removed upon backtracking only. Consequently, when generic dynamisation is used then all the constraints posted so far are active and propagation is duplicated, which decreases overall efficiency.

	SPACE	TIME
ADDVARIABLE(ALLDIFF) *generic dynamisation*	$O(dp)$	$O(dp\sqrt{p})$
ALLDIFF-UPDATE *dynamic all-different*	$O(d+p)$	$O(dp)$

Table 1. Comparison of time and space complexity of adding a variable for the generic dynamisation and for the dynamic implementation of the `alldifferent` constraint.

4.3 Experimental evaluation

We have implemented the dynamic `alldifferent` constraint in SICStus Prolog to compare our approach with the traditional method of dummy variables. To shade off from the complexity of propagation in other constraints, our test examples consist of a single `alldifferent` constraint and several conditional constraints modelling dynamic character of the problem. There are seven basic variables x_1, …, x_7 with the domain {1,..,7} and seven "dependent" variables y_1, …, y_7. Appearance of the variable y_i in the problem depends on the value of the variable x_i; this relation is described using the rules in Table 2.

Set A7	Set B7
$x_1 \leq 4 \Rightarrow$ add variable $y_1 \in \{5,6,\ldots,15\}$	$x_1 \leq 1 \Rightarrow$ add variable $y_1 \in \{5,6,\ldots,10\}$
$x_2 \leq 5 \Rightarrow$ add variable $y_2 \in \{6,7,8\}$	$x_2 \leq 1 \Rightarrow$ add variable $y_2 \in \{6,7,8\}$
$x_3 \leq 5 \Rightarrow$ add variable $y_3 \in \{6,7,8\}$	$x_3 \leq 1 \Rightarrow$ add variable $y_3 \in \{6,7,8\}$
$x_4 \leq 5 \Rightarrow$ add variable $y_4 \in \{6,7,8\}$	$x_4 \leq 1 \Rightarrow$ add variable $y_4 \in \{6,7,8\}$
$x_5 \leq 2 \Rightarrow$ add variable $y_5 \in \{4,5,\ldots,9\}$	$x_5 \leq 2 \Rightarrow$ add variable $y_5 \in \{4,5,\ldots,9\}$
$x_6 \leq 2 \Rightarrow$ add variable $y_6 \in \{4,5,\ldots,9\}$	$x_6 \leq 2 \Rightarrow$ add variable $y_6 \in \{4,5,\ldots,9\}$
$x_7 \leq 2 \Rightarrow$ add variable $y_7 \in \{5,6,\ldots,11\}$	$x_7 \leq 2 \Rightarrow$ add variable $y_7 \in \{5,6,\ldots,9\}$

Table 2. Description of the benchmark problems.

The `alldifferent` constraint is defined over all the variables x_i and y_i that appear in the problem. In the dynamic formulation of the problem, the variable y_i is introduced when the condition of the corresponding rule holds. In the static formulation of the problem, all the variables y_i are introduced at the start with the initial domain $\{1,\ldots,23\}$ and the rules are modelled as a one way propagation from x_i to y_i. The main difference between the set A and the set B is the ratio between the number of variables in the solution and the number of dummy variables. In the test set B7, there will be maximally two variables y_i in the solution but seven dummy variables are necessary. In addition to the test sets A7 and B7 we use smaller sets A6, A5 and B6, B5, where only the first six (five) variables x_i are used (so corresponding rules and dependent variables are removed, while the domains are kept).

Table 3 shows the number of solutions for each of above problems and times to find all the solutions. As we can see there is a significant improvement of time efficiency when the problem is modelled dynamically over the static model with dummy variables. Moreover, when the ratio between the number of necessary variables and the number of dummy variables is decreasing (going from the set A to the set B) then the improvement becomes even more evident.

	# solutions	dummy variables	dynamic version	improvement
A7	5280	6.23 s	4.31 s	44 %
A6	12216	13.06 s	9.20 s	42 %
A5	15612	15.72 s	11.69 s	34 %
B7	9000	13.97 s	8.89 s	57 %
B6	12600	16.85 s	9.14 s	84 %
B5	6390	6.94 s	3.78 s	83 %

Table 3. Comparison of modelling using dummy variables and modelling using the dynamic `alldifferent` constraint. Tests run on Intel Celeron 333 MHz computer

5 Notes on non-monotonic constraints

In the paper we concentrate on dynamisation of monotonic global constraints but there are also many non-monotic global constraints, like the `atleast` constraint. These constraints cannot be dynamised neither using the generic approach presented in Section 2 nor using the semantic-based dynamisation proposed in Section 3. The main obstacle here is that adding new variables to a non-monotonic global constraint might add new solutions to the constraint (see Example 3).

Example 3: Assume that there are variables *a*, *b*, *c* with the domain {1,2,3} and the "global" constraint sum({*a*,*b*,*c*},#=,9), i.e., the sum of the variables equals 9. Visibly, the domain filtering can remove the values 1 and 2 from the domains because these values cannot be part of any solution. Nevertheless, if we extend the set of variables in the sum constraint by adding a new variable *d* with the domain {1,2,3}, then we get the constraint sum({*a*,*b*,*c*,*d*},#=,9). Now, domain filtering does not remove any value from the domains, in particular values 1 and 2 can still be part of some solution (e.g. *a*=1, *b*=2, *c*=3, *d*=3 is a solution satisfying the extended constraint).

There are several ways of using a non-monotonic global constraint in dynamic environment.

First, the constraint is posted only when the set of constrained variables is known. This is a standard way of using global constraints and we already described its disadvantage, namely, domain filtering cannot be applied until all the constrained variables are known.

Second, the non-monotonic global constraints can be made dynamic using incremental addition and removal of values from the domains. When a new variable is added to the non-monotonic constraint, it may violate deletion of values during the domain filtering over a smaller set of variables. Thus we need to restore the original domains of variables and to start domain filtering from scratch which could be rather expensive. Generally speaking, some form of Dynamic CSP is required to handle incremental propagation through non-monotonic constraints. Unfortunately, a simple extension of an existing CSP system cannot do this; the system must be re-programmed from scratch to support Dynamic CSP.

Third, it is possible to approximate the non-monotonic global constraint by a monotonic constraint (see Example 4). This monotonic constraint is then dynamised using the techniques proposed in this paper. As soon as the set of constrained variables is known, propagation through the monotonic approximation is suspended and the non-monotonic global constraint is posted. The advantages of this approach are:

- we can exploit some domain filtering earlier even if the set of constrained variables is not known yet,
- this technique can be easily implemented in existing CSP packages.

Example 4: Assume that we have a non-monotonic constraint sum(*Xs*,#=,*C*), where *Xs* is a list of variables with domains consisting of non-negative integers only and *C* is a constant. This constraint can be approximated by a constraint sum(*Xs*,#=<,*C*), i.e., every solution of sum(*Xs*,#=<,*C*) is also a solution of sum(*Xs*,#=,*C*). Visibly, because the constrained variables are non-negative integers, adding new variables may only increase the sum of variables and thus decrease the upper bounds of domains in the sum(*Xs*,#=<,*C*). The removed values cannot be part of any solution of any extended constraint and, thus, the constraint sum(*Xs*,#=<,*C*) is monotonic.

6 Conclusions

This paper presents a first attempt to formalise the notion of a dynamic global constraint. We restricted ourselves to backtracking based environments where the variables are added and removed in the LIFO order only. This may seem rather restrictive in comparison with say Dynamic CSP but it allows us to implement dynamic global constraints within the existing systems, in particular in Constraint Logic Programming environments. We also showed that this dynamisation pays off as it brings the efficiency advantage over the static approaches when applied to dynamic problems.

The presented ideas may seem straightforward but as far as we know, there are no dynamic global constraints implemented in leading CLP packages like SICStus Prolog, ECLiPSe, or CHIP[3]. Note that the interface to such a dynamic constraint can be very simple: when the user posts the constraint, a variable representing an open-end of the list of constrained variables is returned. Then, the user can easily add new variables to this list when search progresses. This brings not only the advantage of earlier pruning but it also simplifies modelling. In any case, our approach is not worse than waiting until all the variables are known and then finally posting the constraint among them and propagating it. In the highly dynamic problems like planning, our approach is even more beneficial.

[3] The "open" global constraints can be found in C++ based ILOG Solver, though. Our approach provides a more formal view of such constraints.

References

Barták, R. (2000). "Dynamic Constraint Models for Planning and Scheduling Problems." In *New Trends in Constraints.* LNAI 1865. Springer Verlag, 237-255.

Barták, R. (2001). "Dynamic Global Constraints: A First View". In *Proceedings of the CP-AI-OR 2001 Workshop*. 39-49.

Bessiere, Ch., Régin, J.-Ch. (1999). "Enforcing arc consistency on global constraints by solving subproblems on the fly." In *Proceedings of Principles and Practice of Constraint Programming*, 103-117.

Bessiere, Ch. (1999). "Non-binary constraints." In *Proceedings of Principles and Practice of Constraint Programming*, 24-27.

Carlsson M., Ottosson G., Carlsson B. (1997). "An Open-Ended Finite Domain Constraint Solver." In *Proceedings Programming Languages: Implementations, Logics, and Programs*.

Fages, F., Fowler, J., Sola, T. (1995). "A Reactive Constraint Logic Programming Scheme." In *Proceedings of the International Conference on Logic Programming*, 149-163.

Gent, I., Stergiou, K., Walsh, T. (2000). "Decomposable Constraints." In *Artificial Intelligence*, Volume 123, No. 1-2, 133-156.

Mittal, S., Falkemhainer, B. (1990). "Dynamic Constraint Satisfaction Problems." In *Proceedings of AAAI-90*, 25-32.

Nareyek, A. (1999). "Structural Constraint Satisfaction." In *Proceedings of AAAI-99 Workshop on Configuration*.

Nareyek, A. (2000). "AI Planning in a Constraint Programming Framework." In *Proceedings of the Third International Workshop on Communication-Based Systems.*

Pegman, M. (1998). "Short Term Liquid Metal Scheduling." In *Proceedings of Practical Application of Constraint Technology*, 91-100.

Régin, J.-Ch. (1994). "A filtering algorithm for constraints of difference in CSPs." In *Proceedings of AAAI-94*, 362-367.

Régin, J.-Ch. (1996). "Generalised Arc Consistency for Global Cardinality Constraint." In *Proceedings of AAAI-96*, 209-215.

Rossi, F., Dahr, V., Petrie, C. (1990). "On the equivalence of constraint satisfaction problems." In *Proceedings of 9th European Conference on Artificial Intelligence*, 550-556.

Simonis H. (1990). "Trends in Finite Domain Constraint Programming." In *Proceedings of Workshop on Constraint Programming for Decision and Control (CPDC'99).*

Van Der Linden, A.S.J. (2000). *Dynamic Meta-Constraints: An Approach to Dealing with Non-Standard Constraint Satisfaction Problems*. PhD thesis, Oxford Brookes University.

Van Hentenryck, P. (1990). "Incremental Constraint Satisfaction in Logic Programming." In *Proceedings of the International Conference on Logic Programming*, 189-202.

Van Hentenryck, P., Deville, Y. (1991) "The Cardinality Operator: A new Logical Connective for Constraint Logic Programming." In *Proceedings of the International Conference on Logic Programming*, 745-759.