Dynamic Global Constraints in Constraint Logic Programming

Roman Barták*

Charles University, Faculty of Mathematics and Physics
Malostranské náměstí 2/25, Praha 1, Czech Republic
bartak@kti.mff.cuni.cz

Abstract. Constraint Logic Programming (CLP) provides a natural framework for integrating filtering algorithms with backtracking based search. It is straightforward to interleave labelling with introduction of new constraints and variables there, however, this is rarely used due to weak constraint propagation. In this paper, we propose an approach for extending global constraints during labelling, i.e., new variables are added to the constraint depending on value of other variables. This approach preserves the pruning power of global constraints and it allows extending the set of variables constrained by the global constraint on fly.

Keywords: constraint propagation, dynamic problems, global constraints, scheduling

1 Introduction

The traditional approach to solving Constraint Satisfaction Problems (CSP) is static: introduce all the variables, post all the constraints, and finally label the variables. This is useful, if all the objects are know in advance, because we can exploit fully the power of filtering algorithms (domain pruning, constraint propagation or anyhow you call it). However, there exist real-life problems requiring a more dynamic behaviour of the solving engine. 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 [5,16]. 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 [1,7,9]. Approaches like Dynamic Constraint Satisfaction [7] or Structural Constraint Satisfaction [8] were proposed to describe and solve such problems.

* Supported by the Grant Agency of the Czech Republic under the contracts 201/99/D057 and 201/01/0942 and by the project LN00A056 of the Ministry of Education of the Czech Republic.

Constraint Logic Programming seems to be a natural framework for implementing limited dynamic constraint satisfaction. While full Dynamic CSP allows addition/retraction of variables and constraints in arbitrary order, in CLP the new variables and constraints can be added as the search progresses but they can be removed upon backtracking only. This simplifies significantly the implementation using standard Prolog stacks. Nevertheless, this unique feature of interleaving search with constraint posting is rarely used to full extend in CLP. In fact, non-deterministic character of Prolog is used mainly to define disjunction of constraints in separated clauses. As pointed out in [17], this decreases efficiency because such constraints are used to make choices rather than to reduce actively the search space. Moreover, this approach still uses a static view of the problem with labelling starting when all the objects are known. Figure 1 shows how a typical constraint satisfaction problem formulation looks in CLP.

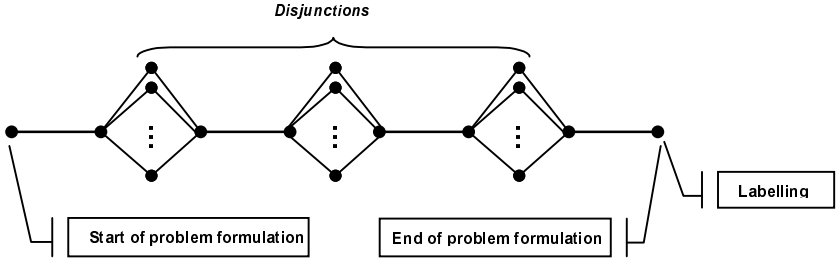

Fig. 1. A typical problem formulation in CLP with disjunction represented using alternative clauses. During the problem formulation (left to right) one of the alternative paths is selected. Another alternative is selected when labelling fails.

Our motivation of using non-deterministic character of Prolog is different from what is described above. In [1], we describe a complex scheduling problem where the static formulation is not appropriate because of huge number of alternatives. The problem there is called process-dependent activities, i.e. the activities whose appearance depends on allocation of other activities. Similar problem was identified in [10] as well; Figure 2 shows the examples.

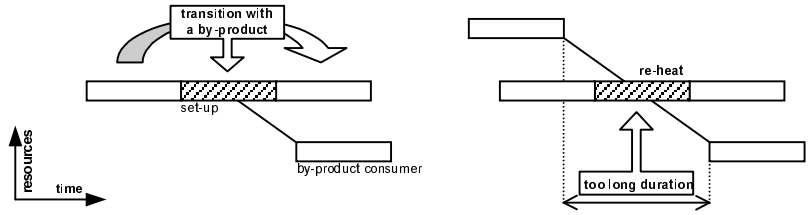

Fig. 2. Gantt charts with set-up (left) and re-heating (right) activities. In both cases a special activity (striped rectangle) is necessary because this activity consumes resource capacity or it is connected to activities in other resources.

In terms of CSP, the process-dependent activities mean that appearance of some variable or constraint depends on value of another variable. In fact, this means that we

cannot post a constraint until we know a value of some variable, i.e., new variables and new constraints might appear during labelling. The obvious solution to solve such problems is to use dummy variables and disjunctive constraints (cardinality operator) before we start labelling. In some sense, the disjunctive constraint is a general representative of various activation and deactivation constraints proposed in [7,15]. Such formulation is appropriate if the alternatives in the problem can be modelled like Figure 1 shows, e.g. in configuration problems. However, some problems, typically in planning, are highly variable and it is impossible to predict which variables and constraints will be used in which combinations. Let's call these problems highly dynamic. Figure 3 shows such a problem formulation in CLP.

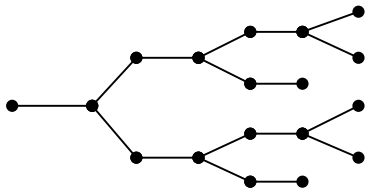

Fig. 3. Formulation of highly dynamic problems forms a tree with many alternatives. Every branch of this tree represents a constraint satisfaction problem with different set of variables and constraints.

The highly dynamic problems cannot be solved using static techniques with dummy (deactivable) variables and constraints due to huge memory consumption of the static representation. To solve such problems we propose to interleave labelling with introduction of new variables and constraints. This helps us to keep reasonable memory consumption but it also weakens constraint propagation because some constraints are introduced late during labelling. Naturally, we prefer to introduce the constraints as soon as possible so they can prune the search space. Constructive disjunction (or in general cardinality operator) allows this for primitive constraints [17].

This paper describes a dynamisation technique for global constraints that allows extending the set of variables constrained by the global constraint on fly. Consequently, the global constraint can be posted even if we do not know yet all the variables that the constraint will involve. Global constraints are necessary for solving large-scale problems thanks to their efficient pruning of variables' domains based on semantic information. Therefore they cannot be fully substituted by generic techniques like cardinality operator. If we interleave labelling with constraint posting, we lost the power of standard global constraints because we must wait until the constrained variables are known before posting the global constraint. Our technique allows posting the global constraint immediately and extending it later by adding new variables. Thus full constraint propagation can be used. In Section 3, we describe a generic dynamisation technique applicable to existing monotonic constraints. This technique suffers from duplicate data structures and computation, therefore we propose to include extendibility into the global constraint directly instead of using meta-programming techniques (Section 4). In Section 5 we compare our approach with existing approaches to solving dynamic problems.

2 Preliminaries and Objectives

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.

We call a complete instantiation $(v_1,\ldots,v_n)$ of the variables, such that $\forall i\ v_i\in D(x_i)$ and $\forall j\ (v_1,\ldots,v_n)\downarrow X(c_j) \in c_j$, a *valid tuple* or *solution*, i.e., the valid tuple is a complete instantiation of the variables satisfying all the constraints. $(v_1,\ldots,v_n)\downarrow X(c_j)$ is a projection of the tuple $(v_1,\ldots,v_n)$ to the variables in $X(c_j)$. 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. 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 $Sol(P')\downarrow X \subseteq Sol(P)$.

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 subproblem in the constraint satisfaction problem. Note, that the constraints in $G(x_1,\ldots,x_l)$ may contain variables other than $x_1,\ldots,x_l$ 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: *alldifferent* global constraint scheme is defined in the following way

$$alldifferent(x_1,\ldots,x_n) = \wedge_{\forall i\neq j}\ x_i\neq x_j$$

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

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

$$atleast(k,y,x_1,\ldots,x_n) = \wedge_{\forall i}\ (b_i \text{ in } \{0,1\})\ \wedge_{\forall i}\ (x_i=y \Leftrightarrow b_i=1) \wedge \Sigma_i\ b_i\geq k$$

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). Note also, that this global constraint scheme must contain at least two variables, *k* and *y*.

[1] It seems that monotonicity feature of the global constraints is closely related to decomposable constraints studied in [6].

Arbitrary n-ary constraint including the global constraints can be decomposed into an equivalent set of binary constraints [13] for which many consistency techniques were designed. However, usually these consistency techniques, like arc consistency, can be extended to n-ary constraints [2]. This could be expensive in general [3] but for global constraints there exist efficient filtering algorithms exploiting semantic of the constraint [11,12,14]. 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 $\forall i\ \forall v_i \in D(x_i)\ \forall j \neq i\ \exists v_j \in D(x_j)$ 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, i.e., $\forall i\ D'(x_i) \subseteq D(x_i) \Rightarrow cons(G(X),D'(X)) \subseteq cons(G(X),D(X))$. 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 $Sol(G(X \cup Y),D(X \cup Y)) \subseteq D'(X \cup Y) \subseteq cons(G(X \cup Y),D(X \cup Y))$, where *cons* is the static A-filtering algorithm for G.

3 Generic Dynamisation Technique

Global constraints are used in constraint programming systems like other constraints, but they 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. As mentioned in Introduction, 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. However, if the global constraint is monotonic we can do better by posting a global constraint over 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

[2] This is a standard way of introducing new consistency algorithms to the constraint engine in CLP [4]. 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.

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 only. As mentioned in Introduction, variables can be removed from the constraint upon backtracking only, 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.

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) - this is done automatically by the underlying constraint engine, 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).

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 revert 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 is used.

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.	$\mathbf{cons(G(X\cup Y),D'(X)\times D(Y)) \subseteq cons(G(X\cup Y),D(X\cup Y))}$	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.	$\mathbf{Sol(G(X\cup Y),D(X\cup Y)) \subseteq cons(G(X\cup Y),D'(X)\times D(Y))}$	9+10

❑

4 Semantic-Based Dynamisation

The dynamisation technique proposed in the previous section suffers from some inefficiencies 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 time complexity,
2) we keep data structures of the old constraint before extension as well as of the new constraint after extension which increases space complexity,
3) if existing interface to the filtering algorithm does not support deactivation (and this is very common) 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 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 Dynamic all-different

The efficient filtering algorithm for the `alldifferent` constraint was proposed in [11]. This 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.

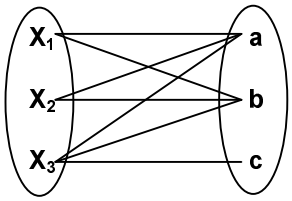

Fig. 4. A value graph for the `alldifferent` constraint with three variables.

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. The description of linear algorithm can be found in [11].

Naturally, the constraint systems consist of more than 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 a domain of a 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 proposed in [11] uses a function MATCHINGCOVERINGX(G, M_1, M_2), which computes a matching M_2, which covers 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 [11] (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 causes such reduction has already failed). The algorithm gets the value graph G, the 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. Remind 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.

The 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∪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.

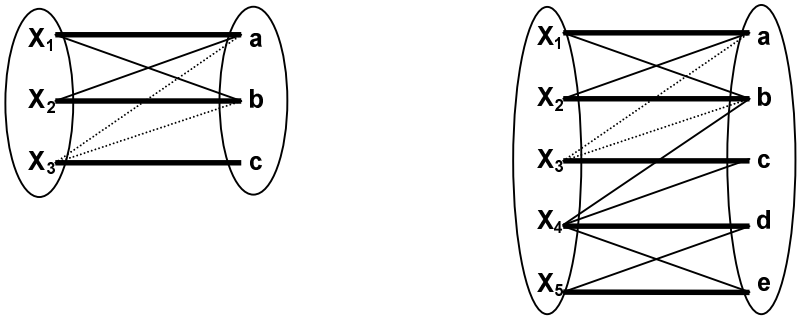

Fig. 5. Edges deleted (dotted edges) from the original value graph (left) do not belong to the extended value graph (right). Bold edges are the maximum matching edges.

After extending the original value graph by adding new edges, the former maximum matching does not cover the extended set X∪Y of variables. We can use the function MATCHINGCOVERINGX to extend the matching which covers X to a matching which covers X∪Y. If there exists a maximum matching which covers X∪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 results from [11] combined with monotonicity feature of the `alldifferent` constraint.

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 [11].

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).

Space complexity. To represent the `alldifferent` constraint we need keeping its value graph. The graph can be represented by the set of its edges so the space complexity is $O(m)$ i.e., $O(pd)$. 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, space complexity of adding a new variable is $O(pd)$.

In case of dynamic filtering algorithm, only new edges are added to the value graph when new variable arrives. These edges can only connect the new variable with the values, so maximal number of new edges is d. Thus, space complexity is $O(d)$, p-times better than generic dynamisation.

Time complexity. According to [11], 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 new constraint).

Time complexity of REMOVEEDGESFROMG is $O(m+p+d)$ [11] so we can use $O(pd)$ instead. 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 which covers all the variables) [11]. Note that if we are adding a new variable then exactly one edge is missing in the maximal covering - the edge connecting the new variable with some value. Consequently, time complexity of MATCHINGCOVERINGX called in ALLDIFF-UPDATE is $O(m)$, that is $O(pd)$. Together, time complexity of adding a new variable using ALLDIFF-UPDATE is $O(pd)$, $\sqrt{p}$-times better than generic dynamisation.

However, note that if we know the variables in advance, 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 $O(dp^2)$. Our algorithm 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 $O(pd)$ [11], 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*	$O(pd)$	$O(dp\sqrt{p})$
ALLDIFF-UPDATE *dynamic all-different*	$O(d)$	$O(dp)$

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

5 Related Works

Solving dynamic constraint satisfaction problems is a hot topic of research in CP, however, not a new one. In 1990 Mittal and Falkenhainer [7] 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. In Introduction we argued that this concept couldn't be applied if the number of model alternatives is very large and a huge number of dummy variables and activity constraints is necessary like in planning [9]. Structural Constraint Satisfaction [8] is an attempt to solve such problems and we believe that standard CLP where labelling is interleaved with constraint posting can serve as a good tool too. Note that alternatives can be modelled using disjunctive constraints but as pointed out by Hentenryck and Deville [17], dynamic models of disjunctive constraints in CLP lost propagation power. Therefore a cardinality operator was proposed in [17] and similar idea can be found in Dynamic Meta-Constraints [15] as well. In our dynamisation of global constraints we are using the same motivation behind cardinality operator: use the constraint as soon as possible to prune domains rather than to wait until a choice is done elsewhere. Note also, that opposite to other approaches to solving dynamic problems our approach does not require the problem to be fully formulated in advance. Thus we can solve highly dynamic problems where the problem formulation forms a search tree with many alternatives [1,8].

Till now we described the approaches for solving problems with internal dynamicity, i.e., the model changes (new variables and constraints are added) as the search progresses. However, there also exists another very important class of dynamic problems where the model is changed by external intervention. These so called incremental, reactive, or on-line problems are studied in [5,16] and it should be said that our approach is not designed to cope with these problems.

6 Conclusions

In this paper, we introduced a concept of dynamic global constraints. The basic idea is to use propagation power of global constraints even if the set of constrained variables is not known yet. Thus we can solve better highly dynamic problems which cannot be modelled statically. We proposed a generic dynamisation technique that can be

applied to arbitrary monotonic global constraint. However, this technique suffers from some efficiency problems and therefore we argue for including dynamic behaviour directly into the filtering algorithm. As an example we presented a dynamic version of the filtering algorithm for the alldifferent constraint.

Future work will include the dynamisation of other global constraints and developing a technique for dynamisation of non-monotonic constraints using special dummy variables.

References

1. Barták, R. Dynamic Constraint Models for Planning and Scheduling Problems. In New Trends in Constraints (Papers from the Joint ERCIM/CompulogNet Workshop, Cyprus, October 25-27, 1999), LNAI 1865, Springer Verlag (2000).
2. Bessiere, Ch., Régin, J.-Ch.: Enforcing arc consistency on global constraints by solving subproblems on the fly. In Proceedings of Principles and Practice of Constraint Programming, pp. 103-117, Alexandria VA, (1999)
3. Bessiere, Ch.: Non-binary constraints. In Proceedings of Principles and Practice of Constraint Programming, pp. 24-27, Alexandria VA, (1999)
4. Carlsson M., Ottosson G., Carlsson B.: An Open-Ended Finite Domain Constraint Solver. In Proc. Programming Languages: Implementations, Logics, and Programs (1997).
5. Fages, F., Fowler, J., Sola, T.: A Reactive Constraint Logic Programming Scheme. In Proceedings of the International Conference on Logic Programming, pp. 149-163 (1995).
6. Gent, I., Stergiou, K., Walsh, T.: Decomposable Constraints. In Artificial Intelligence, Volume 123, No. 1-2, pp. 133-156 (2000).
7. Mittal, S., Falkemhainer, B.: Dynamic Constraint Satisfaction Problems. In Proceedings of AAAI'90, pp. 25-32, Boston (1990).
8. Nareyek, A.: Structural Constraint Satisfaction. In Proceedings of AAAI-99 Workshop on Configuration (1999).
9. Nareyek, A.: AI Planning in a Constraint Programming Framework. In Proceedings of the Third International Workshop on Communication-Based Systems (2000).
10. Pegman, M.: Short Term Liquid Metal Scheduling. In Proceedings of Practical Application of Constraint Technology, London, UK (1998).
11. Régin, J.-Ch..: A filtering algorithm for constraints of difference in CSPs. Research report LIRMM 93-068, LIRM, Université Montpellier, France (1993).
12. Régin, J.-Ch..: Generalised Arc Consistency for Global Cardinality Constraint. In Proceedings of AAAI'96, pp. 209-215, Portland (1996).
13. Rossi, F., Dahr, V., Petrie, C.: On the equivalence of constraint satisfaction problems. In Proceedings of European Conference on Artificial Intelligence, Stockholm (1990).
14. Simonis H.: Trends in Finite Domain Constraint Programming. In Proceedings of Workshop on Constraint Programming for Decision and Control (CPDC'99), Gliwice, Poland (1999).
15. Van Der Linden, A.S.J.: Dynamic Meta-Constraints: An Approach to Dealing with Non-Standard Constraint Satisfaction Problems. PhD thesis, Oxford Brookes University (2000).
16. Van Hentenryck, P.: Incremental Constraint Satisfaction in Logic Programming. In Proceedings of the International Conference on Logic Programming, pp. 189-202, Jerusalem (1990).
17. Van Hentenryck, P., Deville, Y.: The Cardinality Operator: A new Logical Connective for Constraint Logic Programming. In Proceedings of the International Conference on Logic Programming, pp. 745-759 (1991).