Proceedings of CSCLP 2007: Annual ERCIM Workshop on Constraint Solving and Constraint Logic Programming

Francois Fages, Francesca Rossi, Sylvain Soliman

Proceedings of CSCLP 2007: Annual ERCIM Workshop on Constraint Solving and Constraint Logic Programming

June 7-8th, 2007 - Rocquencourt

Organization

François Fages – INRIA Rocquencourt, France
Francesca Rossi – University of Padova, Italy
Sylvain Soliman – INRIA Rocquencourt, France

Foreword

Constraints are a natural way to represent knowledge, and constraint programming is a declarative programming paradigm that has been successfully used to express and solve many practical combinatorial optimization problems. Examples of application domains are scheduling, production planning, resource allocation, communication networks, robotics, and bioinformatics.

These proceedings contain the research papers presented at the 12th International Workshop on Constraint Solving and Constraint Logic Programming (CSCLP'07), held on June 7th and 8th 2007, at INRIA Rocquencourt, France. This workshop, open to all, is organized as the twelfth meeting of the working group on Constraints of the European Research Consortium for Informatics and Mathematics (ERCIM). It continues a series of workshops organized since the creation of the working group in 1997, that have led since 2002 to the publication of a series of books entitled "Recent Advances in Constraints" in the Lecture Notes in Artificial Intelligence, edited by Springer-Verlag.

In addition to the contributed papers collected in this volume, two invited talks were given at CSCLP'07, one by Gilles Pesant, Ecole Polytechnique de Montreal, Canada, and one by Jean-Charles Régin, ILOG, France.

The editors would like to take the opportunity to thank all the authors who submitted a paper, as well as the reviewers for their helpful work. CSCLP'07 has been made possible thanks to the support of the European Research Consortium for Informatics and Mathematics (ERCIM), the Institut National de la Recherche en Informatique et Automatique (INRIA) and the Association for Constraint programming (ACP).

Franois Fages, Sylvain Soliman and Francesca Rossi
Organizers of CSCLP'07

Table of Contents

CP(Graph+Map) for Approximate Graph Matching

Yves Deville[1], Grégoire Dooms[2], Stéphane Zampelli[1], Pierre Dupont[1]

[1] Department of Computing Science and Engineering, Université catholique de Louvain, B-1348 Louvain-la-Neuve - Belgium {yde,sz,pdupont}@info.ucl.ac.be
[2] Department of Computer Science, Brown University, Box 1910, Providence, RI 02912, USA gdooms@cs.brown.edu

Abstract. [3] Graph pattern matching is a central application in many fields. In various areas, the structure of the pattern can only be approximated and exact matching is then too accurate. We focus here on approximations declared by the user within the pattern (optional nodes and forbidden arcs), covering graph/subgraph mono/isomorphism problems.
In this paper, we show how the integration of two new domains of computation over countable structures, *graphs* and *maps*, can be used for modeling and solving approximate graph matching as well as many other morphism problems. To achieve this, we introduce map variables where the domain and range can de declared as finite set variables. We describe how are designed such extended maps, realized on top of finite domain and finite set variables, and specific propagators. On top of CP(Graph+Map), we propose a monomorphism constraint suitable for various morphism problems. Experimental results show that our Gecode implementation solves more problem instances than a dedicated C++ algorithm for graph matching.

1 Introduction

Graph pattern matching is a central application in many fields [1]. Many different types of algorithms have been proposed, ranging from general methods to specific algorithms for particular types of graphs. In constraint programming, several authors [2, 3] have shown that graph matching can be formulated as a CSP problem, and argued that constraint programming could be a powerful tool to handle its combinatorial complexity.

In many areas, the structure of the pattern can only be approximated and exact matching is then far too stringent. Approximate matching is a possible solution, and can be handled in several ways. In a first approach, the matching algorithm may allow part of the pattern to mismatch the target graph (e.g. [4–6]). The matching problem can then be stated in a probabilistic framework

[3] A preliminary version of this paper has been presented at the *1st International Workshop on Constraint Programming Beyond Finite Integer Domains*, CP2005, Sitges (Barcelona), Spain, October 1, 2005.

(see, e.g. [7]). In a second approach, the approximations are declared by the user within the pattern, stating which part could be discarded (see, e.g. [8, 9]). This approach is especially useful in fields, such as bioinformatics, where one faces a mixture of precise and imprecise knowledge of the pattern structures. In this approach, which will be followed in this paper, the user is able to choose parts of the pattern open to approximation.

Within the CSP framework, a model for graph isomorphism has been proposed by Sorlin et al. [10], and by Rudolf [3] and Valiente et al. [2] for graph monomorphism. Subgraph isomorphism in the context of the SBDD method for symmetry breaking is shortly described in [11]. We also proposed in [9] a CSP model for approximate graph matching, but without graph and map variables. Our propagators for monomorphism are based on these works. A declarative view of matching has also been proposed in [12] in the context of XML queries.

In constraint programming, two new domains of computation over countable structures have been introduced: graphs and maps. In CP(Graph) [13], graph domain variables, and constraints on these variables are described (see also [14, 15] for similar ideas). CP(Graph) can be used to express and solve combinatorial graph problems modeled as constrained subgraph extraction problems. In CP(Map) (e.g. [16, 17]), map variables are proposed, but the domain and range are limited to ground sets. Such a high level object is useful for modeling problems such as warehouse location.

In this paper, we propose a Map computation domain and show how approximate graph matching can be modeled and solved, within the CSP framework, on top of CP(Graph+Map).

Contributions The main contributions of this work are the following:

- Introduction of map variables, where the domain and range of the mapping are not limited to ground sets, but can be finite set variables; definition of kernel constraints on CP(Map);
- Description of how a CP(Map) extension can be realized on top of finite domain and finite set variables; design of suitable propagators for map kernel constraints;
- A new theorem, based on matching theory, from which an arc-consistency filtering algorithm can be derived for a global map constraint;
- Integration of CP(Graph+Map) in the Gecode environment;
- Definition of a monomorphism constraint, based on CP(Graph+Map), suitable for modeling and solving different classes of matching problems: monomorphism and isomorphism, graph and subgraph matching, exact and approximate matching;
- Experimental results showing that our Gecode implementation solves more problem instances than a C++ algorithm for (exact) graph matching, without introducing any significant time overhead.

The next section introduces the CP(Graph) framework. The introduction of map variables in CP is described in Section 3. Approximate graph matching is

defined in Section 4, and its modeling within CP(Graph+Map) is handled in Section 5. Section 6 describes experimental results, and Section 7 concludes this paper.

2 CP(Graph)

The CP(Graph) computation domain extends the finite domain and finite sets computation domains with graph variables and constraints. Graph variables are variables whose domain ranges over a set of graphs. As with set variables [18, 16], this set of graphs is represented by a graph interval $[\underline{D}(G), \overline{D}(G)]$ where $\underline{D}(G)$, the greatest lower bound (glb) and $\overline{D}(G)$, the least upper bound (lub) are two graphs with $\underline{D}(G)$ a subgraph of $\overline{D}(G)$ (we write $\underline{D}(G) \subseteq \overline{D}(G)$). These two bounds are referred to as the upper and the lower bound. The lower bound $\underline{D}(G)$ is the set of all nodes and arcs which *must* be part of the graph in a solution while the upper bound $\overline{D}(G)$ is the set of all nodes and arcs which could be part of the graph. The domain of a graph variable with $D(G) = [\underline{D}(G), \overline{D}(G)]$ is the set of graphs g with $\underline{D}(G) \subseteq g \subseteq \overline{D}(G)$. Here, g is used to denote a constant graph and G is used to denote a graph variable. This notation is used throughout this paper: in CSP, lowercase letters denote constants and uppercase letters denote domain variables.

Graph variables can be implemented using a dedicated data-structure or translated into set variables. For instance, a graph variable G can be modeled as a set of nodes N and a set of arcs E with an additional constraint enforcing the relation $E \subseteq N \times N$. Whatever the graph variable implementation, two basic constraints $Nodes(G, SN)$ and $Arcs(G, SA)$ allow to access respectively the set of nodes and the set of arcs of the graph variable. To simplify the notation the expression $Nodes(G)$ is used to represent a set variable constrained to be equal to the set of arcs of G. A similar notation is used for arcs.

Various constraints over graphs have been defined; see for instance the cycle [19], tree [20], path [21, 22], or minimum spanning tree [23]. In the remainder of this article, we use the two simple constraints $Subgraph(G_1, G_2)$ (also denoted $G_1 \subseteq G_2$) and $InducedSubgraph(G_1, G_2)$ (also denoted $G_1 \subseteq^* G_2$). $G_1 \subseteq G_2$ holds if G_1 is a subgraph of G_2, its propagator enforces that the lower bound of G_1 is a subgraph of the lower bound of G_2 and that the upper bound of G_1 is a subgraph of the upper bound of G_2. The constraint $G_1 \subseteq^* G_2$ states that G_1 is the node-induced subgraph of G_2. It holds if G_1 is a subgraph of G_2 such that for each arc a of G_2 whose end-nodes are in G_1, a is also in G_1.

3 CP(Map)

The value of a map variable is a mapping from a domain set to a range set. The domain of a map variable is thus a set a set of mappings. Map variables were first introduced in CP in [16] where Gervet defines relation variables. However, the domain and the range of the relations were limited to ground finite sets. Map variables were also introduced as high level type constructors, simplifying

the modeling of combinatorial optimization problems. This was first defined in [17] as a relation or map variable M from set v into a set w, where supersets of v and w must be known. Such map variables are then compiled into OPL. This idea is developed in [24], but the domain and range of a map variable are limited to ground sets. Relation and map variables are also described in [25] as a useful abstraction in constraint modeling. Rules are proposed for refining constraints on these complex variables into constraints on finite domain and finite set variables. Map variables were also introduced in modeling languages such as ALICE [26], REFINE [27] and NP-SPEC [28]. To the best of our knowledge, map variables were not yet introduced directly in a CP language. One challenge is then to extend current CP languages to allow map variables as well as constraints on these variables.

In the remaining of this section, we show how a CP(Map) extension can be realized on top of finite domain and finite set variables.

3.1 The Map domain

We consider the domain of total surjective functions. Given two elements $m_1 : s_1 \rightarrow t_1$ and $m_2 : s_2 \rightarrow t_2$, where s_1, s_2, t_1, t_2 are sets, we have $m_1 \subseteq m_2$ iff $s_1 \subseteq s_2 \wedge t_1 \subseteq t_2 \wedge \forall x \in s_1 : m_1(x) = m_2(x)$. We also have that $m = \text{glb}(m_1, m_2)$ is a map $m : s \rightarrow t$ with $s = \{x \in s_1 \cap s_2 \mid m_1(x) = m_2(x)\}$, $t = \{v \mid \exists x \in s : m_1(x) = v\}$, and $\forall x \in s : m(x) = m_1(x) = m_2(x)$. The lub between two elements m_1, m_2 exists only if $\forall x \in s_1 \cap s_2 : m_1(x) = m_2(x)$. In that case the lub is a map $m : s \rightarrow t$ with $m(x) = m_1(x)$ if $x \in s_1$, and $m(x) = m_2(x)$ if $x \in s_2$, $s = s_1 \cup s_2$, and $t = \{v \mid \exists x \in s : m(x) = v\}$. The domain of total surjective functions is then a meet semi lattice, that is a semi lattice where every pairs of elements has a glb.

3.2 Map variables and kernel constraints

A map variable is declared with the constraint $MapVar(M, S, T)$, where M is the map variable and S, T are finite set variables. The domain of M is all the *total surjective* functions from s to t, where s, t are in the domain of S, T. We call S the *source set* of M, and T the *target set* of M. When M is instantiated (when its domain is a singleton), the source set and the target set of M are ground sets corresponding to the domain and the range of the mapping. As usual, the domain of a set variable S is represented by a set interval $[\underline{D}(S), \overline{D}(S)]$, the set of sets s with $\underline{D}(S) \subseteq s \subseteq \overline{D}(S)$.

Example Let M be a map variable declared in $MapVar(M, S, T)$, with $dom(S) = [\{8\}, \{4, 6, 8\}]$ and $dom(T) = [\{\}, \{1, 2, 4\}]$. A possible instance of M is $\{4 \rightarrow 1, 8 \rightarrow 4\}$. On this instance, $S = \{4, 8\}$, and $T = \{1, 4\}$. Another instance is $M = \{4 \rightarrow 1, 8 \rightarrow 1\}$, $S = \{4, 8\}$, and $T = \{1\}$.

Map variables can be used for defining various kinds of mappings, such as :

- Total functions : $TotalFct(M, S, T) \equiv T' \subseteq T \wedge MapVar(M, S, T')$
- Partial functions : $PartialFct(M, S, T) \equiv T' \subseteq T \wedge S' \subseteq S \wedge MapVar(M, S', T')$

– Bijective function : $BijectFct(M, S, T) \equiv MapVar(M, S, T) \wedge \forall i, j \in S : M(i) \neq M(j)$.

The kernel constraint on a map variable M is the constraint $Map(M, X, V)$, where X and V are finite domain variables. Given a map variable declared with $MapVar(M, S, T)$, the constraint $Map(M, X, V)$ holds when $X \in S \wedge V \in T \wedge M(X) = V$. We also define the constraint $M1 \subseteq M2$.

3.3 Representing Map variables

When a Map variable M is declared by $MapVar(M, S, T)$ with $\overline{D}(S) = \{x_1, ..., x_n\}$, a finite domain (FD) variable is associated to each x_i. This FD variable will be denoted M_{x_i}. In practice, the implementation allocates an array I of n FD variables. It also allocates a dictionary data structure *index* used to store the index in the array of each value of $\overline{D}(S)$ (i.e. $index(x_j) = j$). The initial domain of each FD variable is $\overline{D}(T) \cup \{\perp\}$ where $\perp$ is a special value used to denote the absence of image for this index. An example is provided in Figure 1.

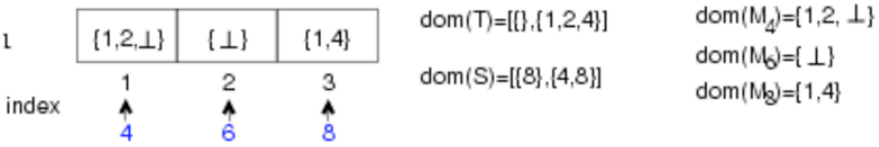


Fig. 1. Implementation of $MapVar(M, S, T)$ (with initial domain $dom(S) = [\{8\}, \{4, 6, 8\}]$ and $dom(T) = [\{\}, \{1, 2, 4\}]$), assuming (other) constraints already achieved some pruning.

The semantics of these FD variables is simple : $M_x = M(x)$. The relationship between the FD variables M_x and the set variables S and T can be stated as follows :

– (1) $S = \{x \mid M_x \neq \perp\}$ (M is total)
– (2) $T = \{v \mid \exists x : M_x = v \neq \perp\}$ (M is surjective)

Given $MapVar(M, S, T)$, the domain of M is the set of total surjective functions $m : s \rightarrow t$ with $s \in D(S)$, $t \in D(T)$, $\forall x \in s : m(x) \in D(M_x)$, and $\forall x \notin s : \perp \in D(M_x)$.

3.4 Propagators

Given two map constraints $MapVar(M1, S1, T1)$ and $MapVar(M2, S2, T2)$ the constraint $M1 \subseteq M2$ is implemented as $S1 \subseteq S2 \wedge T1 \subseteq T2 \wedge \forall x \in S1 : M1_x = M2_x$. The last conjunct can be implemented as a set of propagation rules :

- $x \in \underline{D}(S1) \rightarrow M1_x = M2_x$
- for each $x \in \overline{D}(S1) \setminus \underline{D}(S1) : M1_x \neq M2_x \rightarrow x \notin S1$

The kernel constraint $Map(M, X, V)$ is translated to $X \in S \wedge V \in T \wedge Element(index(X), I, V)$, where S and T are the source and target sets of M, I is the array representing the FD variables M_x, and $index(X)$ is a finite domain obtained by taking the index of each value of the domain of X using the *index* dictionary.

The implementation of $BijectFct(M, S, T)$ is realized through $MapVar(M, S, T) \wedge AllDiffExceptVal(I, \perp) \wedge |S| = |T|$, where I is the array representing the FD variables M_x, and $AllDiffExceptVal$ holds when all the FD variables in I are different when their value is not $\perp$ [29].

Given $MapVar(M, S, T)$, the propagation between M, S and T is based on their relationship described in the previous section, and is achieved by maintaining the following invariants :

- $\overline{D}(S) = \{x \mid D(M_x) \neq \{\perp\}\}$
- $\underline{D}(S) = \{x \in \overline{D}(S) \mid \perp \notin D(M_x)\}$
- $\overline{D}(T) = \{v \mid v \neq \perp \wedge \exists x : v \in D(M_x)\}$
- $\underline{D}(T) \supseteq \{v \mid v \neq \perp \wedge \exists x : D(M_x) = \{v\}\}$

The last invariant is not an equality because when a value is known to be in T, it is not always possible to decide which element in I should be assigned to v.

Propagations rules are then easily derived from these invariants (two rules per invariant) :

$$
\begin{aligned}
M_x = \perp &\rightarrow x \notin \overline{D}(S) \\
x \notin \overline{D}(S) &\rightarrow M_x = \perp \\
x \in \underline{D}(S) &\rightarrow M_x \neq \perp \\
M_x \neq \perp &\rightarrow x \in \underline{D}(S) \\
v \notin \overline{D}(T) \wedge v \neq \perp &\rightarrow v \notin D(M_x) \\
NbOccur(I, v) = 0 \wedge v \neq \perp &\rightarrow v \notin \overline{D}(T) \\
M_x = v \neq \perp &\rightarrow v \in \underline{D}(T) \\
v \in \underline{D}(T) \wedge NbOccur(I, v) = 1 \wedge v \in D(M_x) &\rightarrow M_x = v
\end{aligned}
$$

where $NbOccur(I, v)$ denotes the number of occurences of v in the domains of the FD variables in I. Each of these propagation rule can be implemented in $O(1)$. The implementation of propagators also exploits the cardinality information associated with set variables.

3.5 A global constraint based on matching theory

The above propagators do not prune the M_x FD variables (except the $\perp$ value). We show here how matching theory can be used to design a global constraint

on the relationships (1) and (2) between S, T, and M (defined in Section 3.3), leading to some pruning of the M_x FD variables. We therefore introduce a global contraint $MapVar(M, S, T, MX)$, where MX is the list of the M_x FD variables, which holds if $S = \{x \mid MX_x \neq \bot\}$ and $T = \{v \mid \exists x : MX_x = v \neq \bot\}$

Definition 1. *The* variable-value *graph of a $MapVar(M, S, T)$ constraint is a bipartite graph where the two classes of nodes are the elements of $\overline{D}(S)$ on one side and the elements of $\overline{D}(T)$ plus $\bot$ on the other side. An arc (x, v) is part of the graph iff $v \in D(M_x)$.*

Definition 2. *In a bipartite graph $g = (N_1 \cup N_2, A)$, a* matching *M is a subset of the arcs such that no two arcs share an endpoint : $\forall (u_1, v_1) \neq (u_2, v_2) \in M : u_1 \neq u_2 \wedge v_1 \neq v_2$. A matching M* covers *a set of nodes V, or M is a V-matching of g iff $\forall x \in V : \exists (u, v) \in M : u = x \vee v = x$*

This concept of V-Matching was used by S. Thiel in [30] for the Weighted Partial Alldiff constraint.

The following theorem states the relationship between matching in the bipartite graphs and solutions of the $MapVar$ constraint.

Theorem 1. *Given the constraint $MapVar(M, S, T)$ and its associated variable-value graph g, assuming the constraint is consistent, we have :*

- *(1) Any solution $m : s \rightarrow t$ contains a t-matching of g, and any t-matching can be extended to a solution.*
- *(2) An arc (x, v) belongs to a $\underline{D}(T)$-matching of g,* iff *there exists a solution m with $m(x) = v$*

Proof. (1) The solution m is surjective; every node of t must have at least one incident arc. If we choose one incident arc per node in t, we have a t-matching as m is a function.

Given a t matching, let $m : s \rightarrow t$ be the bijective function corresponding to this matching. Adding arcs to t leads to a surjective function. Let $s' = \underline{D}(S) \cup s$, and $t' = \underline{D}(T) \cup t$. Since the constraint is consistent, $\forall x \in s' \setminus s \; \exists (x, v) \in g : v \neq \bot$, and $\forall v \in t' \setminus t \; \exists (x, v) \in g$. Adding all these arcs leads to a surjective function which is a solution.

(2) ($\Rightarrow$) This is a special case of the second part of (1).

($\Leftarrow$)Let $m : s \rightarrow t$ be a solution with $m(x) = v$. We then have $(x, v) \in g$. By (1), the graph g contains a t-matching M which is also a $\underline{D}(T)$-matching as $\underline{D}(T) \subseteq t$. If $(x, v) \in M$ we are done. Assume $(x, v) \notin M$. Then x is free with respect to M because $M(x) = v$. As $v \in t$, v is covered by M; there is thus a variable node w such that $(w, v) \in M$. Then, x, v, w is an even alternating path starting in a free node. Replacing (w, v) by (x, v) leads to another t-matching, hence a $\underline{D}(T)$-matching of g.

From Theorem 1, an arc-consistency filtering algorithm can be derived : compute the set A of arcs belonging to some $\underline{D}(T)$-matching of the bipartite graph; if $(x, v) \notin A$, remove v from $D(M_x)$. The computation of this set can be

done using techniques such as described in [30], with a complexity of $O(mn)$, where n is the size of T, and m is the number of arcs in the variable-value graph. Our global constraint is also related to the nvalue, range and roots constraints [31, 32].

4 Approximate graph matching

In this section, we define different matching problems ranging from graph monomorphism to approximate subgraph matching. The following definitions apply for directed as well as undirected graphs.

A **graph isomorphism** between a pattern graph $P = (N_p, A_p)$ and a target graph $G = (N, A)$ is a bijective function $f : N_p \rightarrow N$ respecting $(u, v) \in A_p \Leftrightarrow (f(u), f(v)) \in A$. The graph P is isomorphic to G through the function f.

In a **graph monomorphism**, the condition $(u, v) \in A_p \Leftrightarrow (f(u), f(v)) \in A$ is replaced by $(u, v) \in A_p \Rightarrow (f(u), f(v)) \in A$.

Subgraph isomorphism and monomorphism can be defined as graph isomorphism and monomorphism with a subgraph of the target graph. Subgraph isomorphism and monomorphism are known to be NP-complete.

A useful extension is *approximate* subgraph matching, where the pattern graph and the found subgraph in the target graph may differ with respect to their structure [9]. We choose an approach where the approximations are declared by the user in the pattern graph through optional nodes and forbidden arcs.

In graph isomorphism, if two nodes in the pattern are not related by an arc, this absence of arc is an implicit forbidden arc in the matching. It would be interesting to declare which arcs are explicitly *forbidden*, leading to problems between monomorphism and isomorphism.

In graph mono and isomorphism, all the specified nodes and arcs in the patterns must be matched. It would be interesting to allow some of them to be optional. Optional arcs in a graph would only lead to local approximations. We focuss on optional nodes, allowing the pattern to be extended with new nodes. Such nodes are declared *optional* in the pattern graph. Arcs can also be incident to optional nodes. Once an optional node is matched, all its incident arcs to other matched nodes must be matched too. The selected pattern must thus be an induced subgraph of the complete pattern.

In Figure 2, mandatory nodes are represented as filled nodes, and optional nodes are represented as empty nodes. Mandatory arcs are represented with plain line, and arcs incident to optional nodes are represented with dashed lines. Forbidden arcs are represented with a plain line crossed. Matching of node 6 would require the arc $(5, 6)$ to be present in the target. Two matching instances are shown on the right side of Figure 2. The nodes and arcs not selected in the target graph are grey.

A pattern graph with optional nodes and forbidden arcs forms an *approximate pattern graph*, and the corresponding matching is called an *approximate subgraph matching*.[9].

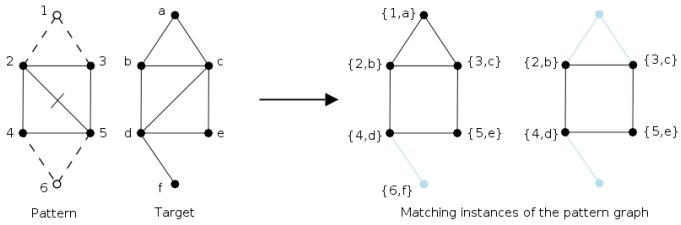


Fig. 2. Example of approximate matching.

Definition 1 *An **approximate pattern graph** is a tuple (N_p, O_p, A_p, F_p) where (N_p, A_p) is a graph, $O_p \subseteq N_p$ is the set of optional nodes and $F_p \subseteq N_p \times N_p$ is the set of forbidden arcs, with $A_p \cap F_p = \emptyset$.*

Definition 2 *An **approximate subgraph matching** between an approximate pattern graph $P = (N_p, O_p, A_p, F_p)$ and a target graph $G = (N, A)$ is a* partial *function $f : N_p \rightarrow N$ such that:*

1. $N_p \setminus O_p \subseteq dom(f)$
2. $\forall\, i, j \in dom(f) : i \neq j \Rightarrow f(i) \neq f(j)$
3. $\forall\, i, j \in dom(f) : (i, j) \in A_p \Rightarrow (f(i), f(j)) \in A$
4. $\forall\, i, j \in dom(f) : (i, j) \in F_p \Rightarrow (f(i), f(j)) \notin A$

The notation $dom(f)$ represents the domain of f. Elements of $dom(f)$ are called the selected nodes of the matching. According to this definition, if $F_p = \emptyset$ the matching is a subgraph monomorphism, and if $F_p = N_p \times N_p \setminus A_p$, the matching is an isomorphism.

5 Modeling approximate graph matching

In this section, we show how CP(Graph+Map) can be used for modeling and solving approximate graph matching as well as many other matching problems.

The problem of graph matching can be stated along three different dimensions:

- monomorphism versus isomorphism;
- graph versus subgraph matching;
- exact versus approximate matching

This leads to 8 different classes of problems. All these problems can be modeled and solved through a single monomorphism constraint on graph domain variables and a map variable.

5.1 The monomorphism constraint

The constraint $Mono(P, G, M)$ holds if P is monomorphic to G through M, where P, G are graph domain variables and M is a map variable with source set $Nodes(P)$ and target set $Nodes(G)$. A $Mono(P, G, M)$ constraint thus implies an implicit $BijectFct(M, Nodes(P), Nodes(G))$.

The $Mono(P, G, M)$ constraint can be defined as follows :

$$Mono(P, G, M) \equiv Biject(M, Nodes(P), Nodes(G)) \\ \wedge (i, j) \in Arcs(P) \Rightarrow (M(i), M(j)) \in Arcs(G)$$

We now show how this constraint can be used to solve the different classes of problems.

5.2 Graph and subgraph mono/iso-morphism

Let p be a pattern graph and g be a target graph. The graphs p and g are ground objects in CP(Graph+Map). Graph monomorphism can easily be modeled as $Mono(p, g, M)$. In a subgraph monomorphism problem, there should exist a monomorphism between p and a subgraph of g, as depicted in Figure 3. The graph G will thus be the matched subgraph of g. The range of M will be the nodes of the matched subgraph of g. Notice that for subgraph matching, it is essential to allow map variables with a finite set variable as target set.

Graph isomorphism can be modeled by two monomorphisms: one between the graphs, and a second between the complementary graphs.

We first introduce a complementary graph constraint $CompGraph(G, Gc)$ which holds if $Nodes(G) = Nodes(Gc) = N$ and $Arcs(Gc) = (N \times N) \setminus Arcs(G)$. We will also use the functional notation $Comp(G)$.

From the definition of graph isomorphism, an isomorphism is a monomorphism with the additional constraint that if an arc does not exist between two pattern nodes, then an arc should not exist through the mapping. This additional constraint states that the mapping should also be a monomorphism between the complementary graphs. In Figure 3, notice that the second $Mono$ constraint could be replaced by $|Arcs(p)| = |Arcs(g)|$, leading to a simpler constraint, but achieving less pruning. It could however be added as a redundant constraint. This also holds for the subgraph isomorphism constraint which can be derived easily, following the same idea than for monomorphism.

5.3 Introducing optional nodes and forbidden arcs

Let us introduce optional nodes in the pattern graph. Let p be the pattern graph with optional nodes, and p_{man} be the subgraph of p induced by the mandatory nodes of p. Approximate graph monomorphism amounts to find an intermediate graph between p_{man} and p which is monomorphic to the target graph. However, between p_{man} and p, only the subgraphs induced by p should be considered. When two optional nodes are selected in the matching, if there is an arc between

	graph	subgraph
monomorphism	$Mono(p, g, M)$	$G \subseteq g \wedge Mono(p, G, M)$
isomorphism	$Mono(p, g, M) \wedge$ $Mono(Comp(p), Comp(g), M)$	$G \subseteq g \wedge Mono(p, G, M) \wedge$ $Mono(Comp(p), Comp(G), M)$

Fig. 3. Exact Matching.

these nodes in pattern graph p, this arc must be considered in the matching, according to our definition of optional nodes. We then obtain the constraints depicted in Figure 4. The symbol $\subseteq^*$ denotes the induced subgraph relation, as defined in Section 2.

Notice that for optional nodes, it is essential to allow map variables with a finite set variable as source set. This easily extends to subgraph monomorphism with optional nodes, as described in Figure 4. In this constraint, the domain of the mapping M will define the selected nodes in the pattern and the range of M will define the selected nodes in the target graph g.

	graph	subgraph
monomorphism	$P \in [p_{man}, p] \wedge$ $P \subseteq^* p \wedge Mono(P, g, M)$	$G \subseteq g \wedge P \in [p_{man}, p] \wedge$ $P \subseteq^* p \wedge Mono(P, G, M)$
isomorphism	$P \in [p_{man}, p] \wedge$ $P \subseteq^* p \wedge Mono(P, g, M) \wedge$ $Mono(Comp(P), Comp(g), M)$	$G \subseteq g \wedge P \in [p_{man}, p] \wedge$ $P \subseteq^* p \wedge Mono(P, G, M) \wedge$ $Mono(Comp(P), Comp(G), M)$

Fig. 4. Matching with optional nodes.

Introducing forbidden arcs to graph monomorphism is not difficult. It is close to the graph isomorphism problem, where the second monomorphism should hold between the graph induced by the forbidden arcs (p_{forb}), and the complementary graph of the target graph, as illustrated in Figure 5. Extension to subgraph matching is straightforward.

graph	subgraph
$Mono(p, g, M) \wedge$ $Mono(p_{forb}, Comp(g), M)$	$G \subseteq g \wedge Mono(p, G, M) \wedge$ $Mono(p_{forb}, Comp(G), M)$

Fig. 5. Matching with forbidden arcs.

5.4 Approximate matching

We now consider the general problem of approximate subgraph matching as defined in the previous section. Given an approximate pattern graph (N_p, O_p, A_p, F_p) where (N_p, A_p) is a graph, $O_p \subseteq N_p$ is the set of optional nodes, and $F_p \subseteq N_p \times N_p$ is the set of forbidden arcs, and a target graph (N, A), we define the following CP(Graph+Map) constants :

- p: the pattern graph (N_p, A_p),
- p_{man}: the subgraph of p induced by the mandatory nodes $N_p \setminus O_p$ of p,
- g: the target graph (N, A),
- p_{forb} : the graph (N_p, F_p) of the forbidden arcs.

The modeling of approximate matching is a combination of subgraph monomorphism with optional nodes, and subgraph isomorphism. But instead of adding a monomorphism constraint on the complementary pattern graph, it is added only on the forbidden arcs.

$$G \subseteq g \wedge P \in [p_{man}, p] \wedge P \subseteq^* p \wedge Mono(P, G, M)$$
$$\wedge\, Nodes(Pc) = Nodes(P) \wedge Pc \subseteq^* p_{forb} \wedge Mono(Pc, Comp(G), M)$$

5.5 Global constraints

A direct implementation of the $Mono(P, G, M)$ constraint based on the definition in the previous section would be very inefficient. A global constraint for

$$(i, j) \in Arcs(P) \Rightarrow (M(i), M(j)) \in Arcs(G)$$

has been designed based on [2, 9], but generalized in the context of CP(Graph+Map). This global constraint is *algorithmically* global as it achieves the same consistency than the original conjunction of constraints, but more efficiently [33].

Redundant constraint, such as proposed in [2, 9] have also been developed to enhance the pruning. Specialized global constraints have also been designed for the different matching families. For instance, in the approximate matching with optional nodes, the $Mono$ propagator is specialized and assumes that a $P \subseteq^* p$ constraint is posted too, allowing the a better pruning. For the isomorphism and for approximate matching with forbidden arcs, a single propagator combining the two $Mono$ propagator is also used, following the ideas developed in [9].

6 Experimental results

This section assesses the performance of the CP(Graph+Map) framework for graph matching. A comparison with a specialized algorithm `vflib` [34] is conducted.

The CP(Graph+Map) framework has been implemented in the `Gecode` system (`http://www.gecode.org`), including graph variables and propagators, map variables and propagators, together with matching propagators.

The graph data set consists of graphs made of different topological structures as explained in [2]. These graphs were generated using the Stanford GraphBase [35], consisting of 1225 undirected instances, and 405 directed instances. The graphs range from 10 to 125 nodes for undirected graphs, and from 10 to 462 for directed graphs.

The experiments consist in performing subgraph monomorphism over the 1225 undirected instances, and subgraph isomorphism over the 405 instances. All solutions are searched. Two frameworks were tested : `vflib` (a C++ specialized algorithm), and our CP(Graph+Map) model. The dedicated algorithm is an improvement of Ullmann's algorithm [36] called `vflib`, described in [37]. Following the methodology used in [2], we put a time limit of five minutes on any given run. A run is called *solved* if it finishes under five minutes or *unsolved* otherwise. All benchmarks were performed on an Intel Xeon 3 Ghz.

Table 6 shows the experimental results. We report the percentage of solved instances (sol.), the percentage of unsolved instances (unsol), the total running time (tot.T), the mean running time (av.T), the mean memory (av.M), the mean running time over instances commonly solved by both approaches (av.T com.), and the mean memory over commonly solved instances (av.M com.).

The CP(Graph+Map) model solves more problem instances than the specialized `vflib` algorithm. This difference is significative for subgraph monomorphism (61% vs. 48%). It is interesting to notice that around 4% of the instances solved by `vflib` were not solved by our CP model. This shows that on some instances, standard algorithms can be better, but that globally, CP(Grap+Map) solves more instances. It is clear that the CP approach consumes more memory. The comparison of the average time is clearly in favour of CP(Graph+Map) as it solves more instances. It is more interesting to compare the mean execution time on the commonly solved instances. This shows that the time overhead induced by the CP approach is minimal on the commonly solved instances : about 9% for monomorphism over undirected graphs and 22% for isomorphism over directed graphs. These experiments show that our CP approach solved significantly more instances than `vflib` for (exact) graph matching, but without introducing any significant time overhead.

7 Conclusion

In this paper, we showed how the integration of two new domains of computation over countable structures, *graphs* and *maps*, can be used for modeling and solving approximate graph matching as well as various other graph matching problems. We already described CP(Graph) in [13]. Maps were already introduced in CP [16], as well as in some modeling languages, but were limited to ground sets for the domain and the range of the map variables. We extended CP(Map) with domain and range of the map variable being finite set variables; we also described

All solutions subgraph monomorphism over undirected graphs (5 min. limit)							
	solved	unsolved	tot.T min	av.T sec	av.M kb	av.T com. sec	av.M com. kb
vflib	48%	51%	3273	160	11.91	4.96	97.6
CP(Graph+Map)	61%	38%	2479	121	9115.46	5.43	8243
All solutions subgraph isomorphism over directed graphs (5 min. limit)							
	solved	unsolved	tot.T min	av.T sec	av.M kb	av.T com. sec	av.M com. kb
vflib	92%	7%	181	26.95	114.28	4.11	4.22
CP(Graph+Map)	96%	3%	109	16.22	2859.85	5.04	2754

Table 1. Comparison for monomorphism and isomorphism problems.

how such extended map variables can be realized on top of finite domain and finite set variables, and we designed propagators for map kernel constraints. We proved a theorem, based on matching theory, from which an arc-consistency filtering algorithm can be derived for a global map constraint.

Approximate matching is based on our work in [9] where approximations are declared by the user within the pattern, stating which part could be discarded (optional nodes and arcs), and also allowing intermediate matching problems from monomorphism to isomorphism through the definition of forbidden arcs.

A monomorphism constraint, defined on graph and map variables has been designed and was shown to be suitable for modeling and solving matching problems with any combination of the following properties : monomorphism or isomorphism, graph or subgraph matching, exact or approximate matching. Graph matching can also be easily integrated within various graph analysis problems, such as constrained path finding [13].

The CP(Graph+Map) framework has been implemented in the `Gecode` system (`http://www.gecode.org`), and is available at `http://cpgraph.info.ucl.ac.be`. Experimental results showed that the CP(Graph+Map) framework for graph matching solves more problem instances than a specialized C++ Ullman (exact) matching algorithm, while also offering approximate matching.

Future work includes the definition of consistency for map variables, the analysis of the pruning induced by the global constraint based on matching theory, the design of a more efficient algorithm (in $O(\sqrt{m}n)$) for this global constraint, the application of our graph matching approach on biochemical network analysis, and the extension of graph matching to other graph comparison problems such as subgraph bisimulation [38].

References

1. Conte, D., Foggia, P., Sansone, C., Vento, M.: Thirty years of graph matching in pattern recognition. IJPRAI **18** (2004) 265–298
2. Larrosa, J., Valiente, G.: Constraint satisfaction algorithms for graph pattern matching. Mathematical. Structures in Comp. Sci. **12** (2002) 403–422

3. Rudolf, M.: Utilizing constraint satisfaction techniques for efficient graph pattern matching. In Ehrig, H., Engels, G., Kreowski, H.J., Rozenberg, G., eds.: TAGT. Volume 1764 of Lecture Notes in Computer Science., Springer (1998) 238–251
4. Wang, J.T.L., Zhang, K., Chirn, G.W.: Algorithms for approximate graph matching. Inf. Sci. Inf. Comput. Sci. **82** (1995) 45–74
5. Messmer, B.T., Bunke, H.: A new algorithm for error-tolerant subgraph isomorphism detection. IEEE Trans. Pattern Anal. Mach. Intell. **20** (1998) 493–504
6. DePiero, F., Krout, D.: An algorithm using length-r paths to approximate subgraph isomorphism. Pattern Recogn. Lett. **24** (2003) 33–46
7. Robles-Kelly, A., Hancock, E.: Graph edit distance from spectral seriation. IEEE Transactions on Pattern Analysis and Machine Intelligence **27-3** (2005) 365–378
8. Giugno, R., Shasha, D.: Graphgrep: A fast and universal method for querying graphs. In: ICPR (2). (2002) 112–115
9. Zampelli, S., Deville, Y., Dupont, P.: Approximate constrained subgraph matching. In Verlag, S., ed.: International Conference on Principles and Practice of Constraint Programming. (2005) 832–836
10. Sorlin, S., Solnon, C.: A global constraint for graph isomorphism problems. In Régin, J.C., Rueher, M., eds.: CPAIOR. Volume 3011 of Lecture Notes in Computer Science., Springer (2004) 287–302
11. Puget, J.F.: Symmetry breaking revisited. Constraints **10** (2005) 23–46
12. Mamoulis, N., Stergiou, K.: Constraint satisfaction in semi-structured data graphs. In Wallace, M., ed.: CP. Volume 3258 of Lecture Notes in Computer Science., Springer (2004) 393–407
13. Dooms, G., Deville, Y., Dupont, P.: Cp(graph): Introducing a graph computation domain in constraint programming. In Verlag, S., ed.: International Conference on Principles and Practice of Constraint Programming. (2005) 211–225
14. Gervet, C.: New structures of symbolic constraint objects: sets and graphs. In: Third Workshop on Constraint Logic Programming (WCLP'93), Marseille (1993)
15. Chabrier, A., Danna, E., Pape, C.L., Perron, L.: Solving a network design problem. Annals of Operations Research **130** (2004) 217–239
16. Gervet, C.: Interval propagation to reason about sets: Definition and implementation of a practical language. Constraints **1** (1997) 191–244
17. Flener, P., Hnich, B., Kiziltan, Z.: Compiling high-level type constructors in constraint programming. In: PADL '01: Proceedings of the Third International Symposium on Practical Aspects of Declarative Languages, London, UK, Springer-Verlag (2001) 229–244
18. Puget, J.F.: Pecos a high level constraint programming language. In: Proceedings of Spicis 92. (1992)
19. Beldiceanu, N., Contjean, E.: Introducing global constraints in CHIP. Mathematical and Computer Modelling **12** (1994) 97–123
20. Beldiceanu, N., Flener, P., Lorca, X.: The tree constraint. In: CPAIOR. (2005) 64–78
21. Sellmann, M.: Cost-based filtering for shorter path constraints. In: Proceedings of the 9th International Conference on Principles and Practise of Constraint Programming (CP). Volume LNCS 2833., Springer-Verlag (2003) 694–708
22. Cambazard, H., Bourreau, E.: Conception d'une contrainte globale de chemin. In: 10e Journées nationales sur la résolution pratique de problèmes NP-complets (JNPC'04). (2004) 107–121
23. Dooms, G., Katriel, I.: The minimum spanning tree constraint. In: Proceedings of the 12th International Conference on Principles and Practice of Constraint Programming. (2006) 152–166

24. Hnich, B.: Function variables for Constraint Programming. PhD thesis, Uppsala University, Department of Information Science (2003)
25. Frisch, A.M., Jefferson, C., Hernandez, B.M., Miguel, I.: The rules of constraint modelling. In: Proceedings of IJCAI 2005. (2005)
26. Lauriere, J.L.: A language and a program for stating and solving combinatorial problems. Artificial Intelligence **10** (1978) 29–128
27. Smith, D.: Structure and design of global search algorithms. Technical Report Tech. Report KES.U.87.12, Kestrel Institute, Palo Alto, Calif. (1987)
28. Cadoli, M., Palopoli, L., Schaerf, A., Vasile, D.: NP-SPEC: An executable specification language for solving all problems in NP. Lecture Notes in Computer Science **1551** (1999) 16–30
29. Beldiceanu, N.: Global constraints as graph properties on structured network of elementary constraints of the same type. Technical Report T2000/01, SICS (2000)
30. Thiel, S.: Efficient Algorithms for Constraint Propagation and for Processing Tree Descriptions. PhD thesis, University of Saarbrucken (2004)
31. Bessière, C., Hebrard, E., Hnich, B., Kiziltan, Z., Walsh, T.: Filtering algorithms for the nvalue constraint. In: CPAIOR. (2005) 79–93
32. Bessière, C., Hebrard, E., Hnich, B., Kiziltan, Z., Walsh, T.: The range and roots constraints: Specifying counting and occurrence problems. In: IJCAI. (2005) 60–65
33. Bessière, C., Van Hentenryck, P.: To be or not to be ... a global constraint. In: Proceedings of the 9th International Conference on Principles and Practise of Constraint Programming (CP). Volume LNCS 2833., Springer-Verlag (2003) 789–794
34. Foggia, P., Sansone, C., Vento, M.: An improved algorithm for matching large graphs. In http://amalfi.dis.unina.it/graph/db/vflib 2.0/doc/vflib.html, ed.: 3rd IAPR-TC15 Workshop on Graph-based Representations. (2001)
35. Knuth, D.E.: The Stanford GraphBase. A Platform for Combinatorial Computing. ACM, NY (1993)
36. Ullmann, J.R.: An algorithm for subgraph isomorphism. J. ACM **23** (1976) 31–42
37. Cordella, L.P., Foggia, P., Sansone, C., Vento, M.: Performance evaluation of the vf graph matching algorithm. In: ICIAP, IEEE Computer Society (1999) 1172–1177
38. Dovier, A., Piazza, C.: The subgraph bisimulation problem. IEEE Transaction on Knowledge and Data Engineering **15** (2003) 1055–1056

Spontaneous Simplifications in Graphs of Partitions

Daniel Goossens

LIASD, Universite Paris 8, 2 rue de la Liberte, F-93526 Saint-Denis, France.
email: goo@ai.univ-paris8.fr

Abstract. This paper presents a simplifier for graphs of partitions, a constraints representation formalism which may represent arbitrary boolean expressions through variable dependencies. The main advantage of using these graphs is that they imply systems of modulo 2 linear equations which are immediately extractible. Mixing gaussian elimination and boolean propagations allows the simplifier to suppress some contradictory nodes and merge some equivalent nodes independently of a particular problem to solve. Two theorems are proved that restrict the nodes to test for contradiction or equivalence. Experiments show that the simplifier is a lightweight bookkeeping mechanism.

1 Introduction

The role of simplification is to transform some problem instance so as to make it tractable or solve it more quickly. Spontaneous simplifications are those simplifications that are potentially useful for all problems. They preserve equivalence with the original base of constraints, not only satisfiability. They may thus spend more time that the solving of a particular problem instance since they are performed only once on constraints that are shared by future problems to solve. For instance, spontaneous simplifications are expected to keep a knowledge base free from some redundancies, like contradictory or equivalent expressions. These simplifications *repair* the graphs of constraints, as they are incrementally modified. In other words, they perform deductions rather than satisfiability tests. In graphs of partitions, variables and constraints are encoded as nodes and hyperlinks (partitions). The deductions spontaneously find equivalent or contradictory nodes rather than verify them.

They constrast with preprocessing techniques, which simplify problem instances, where all transformations are allowed provided they help to solve a particular instance. Preprocessing has been used in the context of SAT solving to search for "frozen" variables, whose value is the same in all assignments, by testing every node with some propagation algorithm, or to extract implications, equivalences and other boolean gates from cnf formulas [5]. Preprocessing is worth only if the time spent to simplify and solve a problem is less than the time to solve the problem. Some efficient and powerful preprocessors are based on the elimination of boolean variables by resolution. Different inferences are

made depending on the choice of the variable to eliminate. The choice order is irrelevant for proving unconsistency of a formula. The Non-increasing Variable Elimination Resolution (NiVER) preprocessor of cnf [11] randomly chooses variables to eliminate and prohibits those that increase the number of literals. The resulting simplifications are relative to the current problem instance to solve. On another hand, detecting contradictory variables or equivalent variables with variable elimination would require to explore all possible orderings of the sequence of eliminated variables. The present work may be seen as a proposal to control such simplifications with generalistic goals like suppressing contradictory nodes and merging equivalent nodes.

The graph of partitions is an early data structure in artificial intelligence [9]. It allows to represent arbitrary boolean expressions through variables dependencies and it implies a system of modulo 2 linear equations [3]. The simplifier mixes gaussian elimination on the linear system and propagations of boolean values along the partitions.

Solving modulo 2 linear equations, or parity reasoning, has been exploited in propositional reasoning. [8] bases satisfiability testing on the Boolean ring algebraic structure, where OR ($\vee$) is replaced by XOR ($\oplus$). Linear equations modulo 2, or affine formulas, are one of the polynomial classes for the SAT problem, identified by [10]. [12] and [6] successfully solved new challenges on the satisfiability problem in propositional logic by deducing XOR-clauses from standard OR-clauses and reasoning from them. [1] integrated gaussian elimination and DPLL in a single algorithm which operates on mixings of OR-clauses (cnf) and XOR-clauses. [4] extensively studied the combination of linear modulo 2 reasoning with the cnf reasoning of the DPLL procedure. What makes these approaches specialized is that XOR-clauses may be arbitrarily hidden in the cnf formalism, and rarely under the form expected by a given extracting algorithm.

The originality of graphs of partitions is that modulo 2 linear equations are immediately available from partitions. In other boolean formats, affine relations need to be extracted or provided separately (as XOR-clauses). In [13], for instance, affine formulas are used to approximate boolean knowledge and methods are provided for extracting affine relations from general boolean relations.

The main contributions of the paper are the simplifier itself, a formal proof of propositions which restrict the number of nodes to test for contradiction at each modification of the graph, replacing an $O(n^2)$ obvious strategy by an $O(n)$ one based on the dilemma rule, and experiments which show that the spontaneous simplifier is rapid enough to be used as an interactive tool.

The paper is organized as follows. Section 2 presents graphs of partitions and their linear component. Sections 3 and 4 present the propagation mechanisms of the simplifier and the bi-propagation algorithm. Section 5 presents experimental results.

2 Graphs of Partitions

Graphs of partitions are oriented hypergraphs. The hyperlinks are called partitions and are sets of nodes of the graph. Each partition contains a distinguished head node.

Definition 1. *A* partition *is a couple* $< t, \{f_1 \dots f_n\} >$*, where* t *and* $f_1 \dots f_n$ *are nodes.* t *is the* head *of the partition and* $f_1 \dots f_n$ *are its* leaves.

Definition 2. *A* graph of partitions *is a couple* $< N, L >$*.* N *is a set of nodes.* L *is a set of partitions whose nodes are in* N*.*

Partitions may be interpreted as set constraints or boolean variables. When interpreted as sets, the head of a partition is a set and its leaves are the elements of a partition of this set. In the boolean interpretation, nodes are boolean variables and partitions are constraints on them.

2.1 Boolean Interpretation

Under the boolean interpretation, nodes are boolean variables and partitions are constraints. For clarity, we restrict first to bipartitions and generalize to arbitrary partitions. the definitions become :

Definition 3. *A* bipartition $< x, \{y, z\} >$ *is a boolean constraint* $(x \leftrightarrow (y \vee z)) \wedge \neg(y \wedge z)$ *over the boolean variables* x, y, z*.*

A bipartition $< x, \{y, z\} >$ is abbreviated as $x = y + z$. The expression $y + z$ is the XOR operation $y \oplus z$ augmented with the constraint $\neg(y \wedge z)$. Each bipartition $x = y + z$ is thus equivalent to the cnf $(\overline{x} \vee y \vee z)(x \vee \overline{y})(x \vee \overline{z})(\overline{y} \vee \overline{z})$ and to the CSP constraint $< x, y, z > \in \{< 0, 0, 0 >, < 1, 1, 0 >, < 1, 0, 1 >\}$.

This interpretation generalizes to arbitrary partitions $< t, \{f_1 \dots f_n\} >$, abbreviated as $t = f_1 + \ldots + f_n$. The f_i are mutually disjoint and t is the XOR of the f_i. Thus, a partition is the conjunction of two constraints :

$$t \oplus f_1 \oplus \ldots \oplus f_n = 0$$

$$\forall i \in [1, n]\ \forall j \in [1, n]\ (i \neq j \rightarrow \neg(f_i \wedge f_j))$$

Among the 2^{n+1} valuations of variables $t, f_1 \dots f_n$, the partition $t = f_1 + \ldots + f_n$ is a constraint which allows only $n + 1$ valuations : one where all variables are false and those where t and a single f_i are true.

Definition 4. *A* graph of partitions *is a conjunction of partitions.*

Definition 5. *A* valuation *of a graph of partitions* G *is a conjunction of constraints* $x = 0$ *or* $x = 1$*, where* x *is a node of* G*. A* valuated *graph of partitions is a conjunction* $G \wedge Val$*, where* G *is a graphs of partitions and* Val *is a valuation of* G*.*

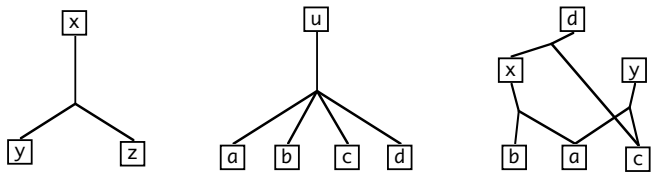


Fig. 1. A graph of partitions is drawn as a directed hypergraph. Hyperlinks represent partitions. The first graph is the bipartition $x = y + z$. x is its head and y and z its leaves. The second graph is the partition $u = a + b + c + d$. The third graph is the conjunction of the bipartitions $x = a + b$, $y = a + c$, $d = x + c$.

2.2 The Linear Component

Every partition implies a modulo 2 linear equation and every graph of partitions implies a system of modulo 2 linear equations.

Proposition 1. *Every partition $t = f_1 + \ldots + f_n$ implies the modulo 2 linear equation*

$$t \oplus f_1 \oplus \ldots \oplus f_n = 0$$

Definition 6. *Let $G = p_1 \wedge \ldots \wedge p_n$ be a graph of partitions. For each p_i, let e_i be the linear equation implied by p_i. $(e_1 \wedge \ldots \wedge e_n)$ is the* modulo 2 linear system implied by G. *It is also noted as $\{e_1, \ldots, e_n\}$.*

The set of equations deducible from a graph is a vector space over the field F_2. Addition is $\oplus$ and multiplication is $\wedge$. The system implied by a graph is solved by gaussian elimination. When the graph is not valuated, the system it implies is not contradictory. Gaussian elimination then defines some nodes as linear combinations of others. It may detect some contradictory nodes (null vector) and pairs of equivalent or disequivalent nodes. When the graph is valuated, gaussian elimination detects linear contradictions and extends the current valuation by unit propagation on equations with a single non-valuated variable.

Every modulo 2 linear equation may be written as $V = 0$ or $V \oplus 1 = 0$, where V is a sum of variables. Let $e1 : V1 = 0$ and $e2 : V2 = 0$ be two equations. The operation $e1 \oplus e2$ which returns the equation $V1 \oplus V2 = 0$ is the only basic deduction operation of gaussian elimination.

If $R = \{e_1, \ldots, e_k\}$ is a set of equations, the equation $(e_1 \oplus \ldots \oplus e_k)$ is noted $\oplus R$.

Let $S = \{e_1, \ldots, e_n\}$ be the system of n equations implied by a graph. The set of equations deducible from S is noted $E(S)$. $E(S) = \{\oplus R \ / \ R \subseteq S\}$.

$degree(x, R)$ is the number of occurrences of variable x in the equation set R. The fact that variable x appears in R is noted $x \in R$.

If a subset R of S contains only variables of even degree in R ($\exists R \subseteq S$ ($\forall x \in R$ $degree(x, R)$ *is even*)), then $\oplus R$ only contains constants. After simplification, $\oplus R$ is either $0 = 0$ or $0 = 1$. If $\oplus R$ is the equation $0 = 1$, the system is

contradictory. Gaussian elimination then transforms S in a solved system which contains the equation $0 = 1$ (contradiction).

Otherwise, $\forall R \subseteq S$ ($\exists x \in R$ *degree*(x, R) *is odd*). Every equation e of $E(S)$ then contains at least one variable and the system is not contradictory. Gaussian elimination transforms S into a solved system $S_r = \{r_1 = T_1, \ldots, r_n = T_n\}$ equivalent to S (that is, $E(S) = E(S_r)$). Each T_i is either 0, V_i or $V_i \oplus 1$, where V_i is a sum of variables. The variables in the V_i are a basis of the vector space. The r_i are the *defined variables*, vectors with coordinates in the basis. Each r_i appears in a single equation. As $E(S_r)$ is a vector space, the mapping f which maps each $R \subseteq S_r$ into the subset of $\{r_1, \ldots, r_n\}$ which contains the defined variables of equation $\oplus R$ is a bijection. It follows that gaussian elimination is complete for the detection of contradictory nodes and pairs of equivalent nodes on the linear system implied by a graph of partitions.

Let S be a non contradictory system of modulo 2 linear equations and S_r the equivalent system, solved by gaussian elimination.

Proposition 2. *S_r contains all the equations $x = 0$ or $x = 1$ deducible from S.*

Proof. Since the mapping f between $E(S_r)$ and the set of subsets of $\{r_1, \ldots, r_n\}$ is bijective, every equation of $E(S_r)$ not in S_r is a sum of more than one equations of S_r and contains more than one variable. So, all the equations of $E(S_r)$ which contain a single variable are in S_r. □

Proposition 3. *If a linear equation $x = y$ of $E(S_r)$ is not in S_r, then S_r contains a pair $\{x = V, y = V\}$, where V is a sum of variables.*
If a linear equation $x = y \oplus 1$ (that is, $x \neq y$) of $E(S_r)$ is not in S_r, then S_r contains a pair $\{x = V, y = V \oplus 1\}$, where V is a sum of variables.

Proof. Since f is a bijection, every equation of $E(S_r)$ contains at least a defined variable of S_r. If $E(S_r)$ contains an equation e of the form $x = y$ or $x = y \oplus 1$, at least one of x and y is a defined variable. If e contains a single defined variable, it is in S_r. Otherwise, x and y are two defined variables and $e = e_x \oplus e_y$, where e_x and e_y are the equations $x = V_x$ and $y = V_y$ of S_r. If e is $x = y$, then $V_x \oplus V_y = 0$, thus $V_x = V_y$. If e is $x = y \oplus 1$, then $V_x \oplus V_y = 1$, thus $V_y = V_x \oplus 1$. □

Example 1. In Fig. 2 of Subsect. 4.1, linear equations are represented either as partitions or as grey hyperlinks with a circled center. The defined variable of each equation is indicated by a small dot. On the left graph, for instance, the basis is the set $\{n, 1, 2, 3, 4\}$ and the dotted nodes $\{x, y, 5, 6, 7\}$ are linear combinations of this basis.

3 Propagations of Boolean Values

Definition 7. *Let G be a graph of partitions, x and y two nodes of G and v and w two boolean values.* $(x = v)$ propagates $(y = w)$ *in G, noted* $(G \wedge (x = v)) \vdash (y = w)$*, if and only if $y = w$ is deduced by some propagation algorithm from $G \wedge (x = v)$.*

Notations : when there is no ambiguity on which graph G is referred to, $(x = v) \vdash (y = w)$, or $x = v \vdash y = w$, means $(G \wedge (x = v)) \vdash (y = w)$, noted also $(x = v) \vdash (y = w)$ *in* G. The sequence $x = a \vdash y = b \vdash z = c \ldots$ means $(x = a \vdash y = b) \wedge (y = b \vdash z = c) \ldots$

3.1 The Divergent Propagation

The divergent propagation is limited to the deduction of inclusion and disjunction relations. It is specific to graphs of partitions. Its separation from a more powerful propagation is justified by Props. 9 and 10 : When a graph is locally modified, by merging two nodes or adding a partition, it is not needed to test every node in the graph to detect contradictory nodes by divergent propagation.

Each partition $p =< t, \{f_1 \ldots f_n\} >$ implies $f_i \rightarrow t$ for each leave f_i of p, and the f_i are mutually disjoint. The divergent propagation is based on these implications. It is noted $\vdash_{div}$.

Definition 8 defines the predicate $(G \wedge (x = v)) \vdash_{div} (y = w)$ which is true if and only if $x = v$ propagates $y = w$.

Definition 8 (divergent propagation). *Let G be a graph,*

1- $\forall x$ *node of* G, $\forall v \in \{0,1\}$, $(x = v \vdash_{div} x = v)$
2- $\forall p =< t, F >$ *partition of* G,
$(\forall f \in F,\ (f = 1 \vdash_{div} t = 1)\ \wedge (t = 0 \vdash_{div} f = 0))$
$\wedge\ (\forall f \in F\ \forall g \in F\ / f \neq g,\ (f = 1 \vdash_{div} g = 0))$
3- $\forall xyz$ *nodes of* G, $\forall abc \in \{0,1\}^3$
$((x = a \vdash_{div} y = b) \wedge (y = b \vdash_{div} z = c)) \rightarrow (x = a \vdash_{div} z = c)$
$\forall xy$ *nodes of* G, $\forall ab \in \{0,1\}^2, \neg(x = a \vdash_{div} y = b)$ *otherwise.*

The correctness of divergent propagation follows directly from the boolean interpretation of partitions.

Propositions 4 and 5 simply express the restricted nature of divergent propagation. Proposition 4 makes equivalent the fact that x is an ancestor of y in G and the fact that $x = 1$ propagates $y = 1$ and $y = 0$ propagates $x = 0$. Proposition 5 says that $x = 1$ propagates $y = 0$ iff G contains a partition p such that x and y are descendants of two different leaves of p. Informally, the property $ipath(x, y, G)$ (for inclusion path) is true if and only if x is a descendant of y in G :

Definition 9 (ipath). *Let G be a graph,*

$\forall x$ *node of* G, $ipath(x, x, G)$
$\forall p =< t, F >$ *partition of* G, $\forall f \in F$ $ipath(f, t, G)$
$\forall xyz$ *nodes of* G, $(ipath(x, y, G) \wedge ipath(y, z, G)) \rightarrow ipath(x, z, G)$
$\forall xy$ *nodes of* G, $\neg ipath(x, y, G)$ *otherwise.*

Proposition 4. *Let G be a graph.* $\forall xy$ *nodes of* G,
$(ipath(x, y, G) \leftrightarrow (x = 1 \vdash_{div} y = 1)) \wedge (ipath(x, y, G) \leftrightarrow (y = 0 \vdash_{div} x = 0))$

Proof. By case analysis and induction on Defs. 8 and 9. The proofs of the two members of the conjunction are symmetrical.
Only $ipath(x, y, G) \leftrightarrow (x = 1 \vdash_{div} y = 1)$ needs to be proved :

case 1 : $x = y$
$ipath(x, x, G) \leftrightarrow (x = 1 \vdash_{div} x = 1)$
since both members of the equivalence are true.
case 2 : $\exists p =< y, \{x\} \cup F >$ partition of G
$ipath(x, y, G) \leftrightarrow (x = 1 \vdash_{div} y = 1)$
idem.
case 3 : $\exists z\ (ipath(x, z, G) \leftrightarrow (x = 1 \vdash_{div} z = 1)) \wedge$
$(ipath(z, y, G) \leftrightarrow (z = 1 \vdash_{div} y = 1))$ Recurrence Hypothesis
$(ipath(x, z, G) \wedge ipath(z, y, G)) \leftrightarrow ((x = 1 \vdash_{div} z = 1) \wedge (z = 1 \vdash_{div} y = 1))$
Bool. Implic. of Recurrence Hypothesis.
If both members of the equivalence are true :
$ipath(x, z, G) \wedge ipath(z, y, G)$ Hypothesis
1. $ipath(x, y, G)$ Def. 9
$(x = 1 \vdash_{div} z = 1) \wedge (z = 1 \vdash_{div} y = 1)$ Hypothesis
2. $(x = 1 \vdash_{div} y = 1)$ Def. 8
$ipath(x, y, G) \leftrightarrow (x = 1 \vdash_{div} y = 1)$ since 1, 2 true
If both members of the equivalence are false :
$\neg\exists z\ (ipath(x, z, G) \wedge ipath(z, y, G))$ Hypothesis
3. $\neg ipath(x, y, G)$ Def. 9
$\neg((x = 1 \vdash_{div} z = 1) \wedge (z = 1 \vdash_{div} y = 1))$ Hypothesis
4. $\neg(x = 1 \vdash_{div} y = 1)$ Def. 8
$ipath(x, y, G) \leftrightarrow (x = 1 \vdash_{div} y = 1)$ since 3, 4 false
$ipath(x, y, G) \leftrightarrow (x = 1 \vdash_{div} y = 1)$ proved by induction
Otherwise, both members are false and the equivalence is true. □

Proposition 5. *Let G be a graph. $\forall xy$ nodes of G, $(x = 1 \vdash_{div} y = 0) \leftrightarrow$ $(\exists p$ partition of G, $\exists fg$ leaves of p $/ f \neq g$, $ipath(x, f, G) \wedge ipath(y, g, G))$.*

Proof. Implications are proved separately.

implication $\leftarrow$:
$(p$ partition of $G) \wedge (f$ leaf of $p) \wedge (g$ leaf of $p) \wedge f \neq g$
$\wedge\ ipath(x, f, G) \wedge ipath(y, g, G)$ Hypothesis
$x = 1 \vdash_{div} f = 1 \vdash_{div} g = 0 \vdash_{div} y = 0$ prop. 4, def. 8
implication $\rightarrow$:
$x = 1 \vdash_{div} y = 0$ Hypothesis
Two cases of Def. 8 may imply this hypothesis :
case 2 : there exists a partition $p =< t, \{x, y\} \cup F >$ in G,
$ipath(x, x, G) \wedge ipath(y, y, G)$ Def. 9
$\exists p$ partition of G, $\exists fg$ leaves of p $/ f \neq g$,
$ipath(x, f, G) \wedge ipath(y, g, G)$ generalization
case 3 :
Two subcases imply $x = 1 \vdash_{div} y = 0$:

case 3.1 :
1. $(x = 1 \vdash_{div} z = 1) \wedge (z = 1 \vdash_{div} y = 0)$ Hypothesis
proof by recurrence :
2. $(z = 1 \vdash_{div} y = 0) \rightarrow (\exists p \; partition \; of \; G, \; \exists fg \; leaves \; of \; p \; / f \neq g,$
$ipath(z, f, G) \wedge ipath(y, g, G))$ Recurrence Hypothesis
$\exists p \; partition \; of \; G, \; \exists fg \; leaves \; of \; p \; / f \neq g,$
$ipath(z, f, G) \wedge ipath(y, g, G)$ MP, 1 2
3. $(p \; partition \; of \; G) \wedge (f \; leaf \; of \; p) \; \wedge \; (g \; leaf \; of \; p) \; \wedge \; f \neq g$
$\wedge \; ipath(z, f, G) \wedge ipath(y, g, G)$ instanciation
4. $ipath(x, z, G)$ Proposition 4, 1
5. $ipath(x, f, G)$ MP, 3 4
$\exists p \; partition \; of \; G, \; \exists fg \; leaves \; of \; p \; / f \neq g,$
$ipath(x, f, G) \wedge ipath(y, g, G)$ generalization, 3 5
case 3.2 :
$(x = 1 \vdash_{div} z = 0) \wedge (z = 0 \vdash_{div} y = 0)$
This case is symmetrical with case 3.1 □

The Proposition 6 says that whenever the divergent propagation can do an implication, it can do its contraposite. It is only mentioned in Subsect. 3.2 so the proof is omitted.

Proposition 6 (contraposite). *Let G be a graph, x y two nodes of G, v w two boolean values.* $(x = v \vdash_{div} y = w) \leftrightarrow (y = \neg w \vdash_{div} x = \neg v)$.

From Def. 8 and by symmetry of $\leftrightarrow$, two cases need to be proved : $(x = 0 \vdash_{div} y = 0) \leftrightarrow (y = 1 \vdash_{div} x = 1)$ and $(x = 1 \vdash_{div} y = 0) \leftrightarrow (y = 1 \vdash_{div} x = 0)$. Both cases present no difficulty.

3.2 The Convergent Propagation

Each partition $p =< t, \{f_1 \dots f_n\} >$ implies the modulo 2 linear equation $(t \oplus f_1 \oplus \dots \oplus f_n = 0)$. The convergent propagation is based on these implications. The linear system implied by a graph is solved by gaussian elimination. The convergent propagation then does unit propagation on the equations with a single non valuated variable :

Rconv :
$[(x1 = v1) \wedge \dots \wedge (xk = vk) \wedge (x = (x1 \oplus \dots \oplus xk))] \vdash_{conv} (x = (v1 \oplus \dots \oplus vk))$

The convergent propagation does not verify Propositions 5 and 6. To make it verify them, it suffices to memorize each implication $x \rightarrow y$ deduced by rule *Rconv* with a bipartition $y = x + z$, where z is a new node.

Proposition 7. *Let $< G, val >$ be a valuated graph. Let $S = \{e_1 \dots e_n\}$ be the modulo 2 linear system implied by G and including the equations $(n = v)$ of the valuation val. If S implies a linear equation with a single non valuated variable, where Rconv is triggered, then this equation is deduced by gaussian elimination.*

Proof. By completeness of gaussian elimination for the deduction of equations $(x = 0)$ or $(x = 1)$. If x is the only non valuated variable of an equation, it simplifies in $(x = 0)$ or $(x = 1)$. □

4 The Bi-propagation

The objective of bi-propagation is to suppress contradictory nodes n in a graph, which are detected by $(n = 1) \vdash (n = 0)$, without testing every node each time the graph is modified. The symbol $\vdash$ refers to some propagation algorithm.

The bi-propagation is called by the linear reasoning component every time two equivalent nodes are merged or a new partition is added. It does a single test. The test is a specialization of the dilemma rule, restricted to the divergent propagation. When two nodes are merged into a node z, `bi-propagate(z)` is called. When a partition p is added, `bi-propagate(p)` is called :

```
bi-propagate(z) {
   propagate(z, 1)
   propagate(z, 0)
   suppress every node n receiving 0 from both propagations
}

bi-propagate(t = f1 + ... + fn) {
   propagate(t, 1)
   for i=1 to n
      propagate(fi, 0)
   suppress every node n receiving 0 from two propagations
}
```

The `suppress` operation adds the equation $n = 0$ to the linear system and solves it. This will eventually deduce other simplifying equations. The simplifier refers to the solved linear system to effectively modify the graph.

If `propagate` is restricted to the divergent propagation, Props. 9 and 10 express a form of completeness. Let x and y be two nodes of G. Suppose that $G \wedge (x = y)$ contains a contradictory node n, that the divergent propagation detects in $G \wedge (x = y)$ but not in G. Proposition 9 says that n is detectable from x in the graph $G \wedge (x = y)$ by bi-propagation (test $(x = 1) \vdash_{div} (n = 0)$ and $(x = 0) \vdash_{div} (n = 0)$).

In the same way, if p is a partition not in G and n a node of G, it may be that $(n = 1) \vdash_{div} (n = 0)$ in $G \wedge p$ and not in G. Proposition 10 says that n is detectable from p in $G \wedge p$ by bi-propagation.

To prove Prop. 9, the predicate *ipath* (Def. 9) must be related to its models. The models verifying $ipath(a, b, G)$ are sets of partitions $\{p_1, \ldots p_n\}$ of G such that a is a leaf of p_1, b is the head of p_n and for i from 1 to $n - 1$, the head of p_i is a leaf of p_{i+1}.

The proof of Prop. 9 is based on the following principle, which is admitted since it would require a formalisation of substitution : when two nodes x and y of

a graph G are merged (G becomes $G \wedge (x = y)$), if an inclusion path $\{p_1, \ldots p_n\}$ exists (model of $ipath(a, b, G \wedge (x = y))$ which does not contain the merged node $x = y$, then each p_i is also in G and $ipath(a, b, G)$ is deducible :

Proposition 8. *Let G be a graph. $\forall abxy$ nodes of G, $ipath(a, b, G \wedge (x = y)) \wedge \neg(ipath(a, x, G \wedge (x = y)) \wedge ipath(x, b, G \wedge (x = y))) \rightarrow ipath(a, b, G)$*

Proposition 9. *Let G be a graph. Let n be a node of G such that $\neg(n = 1 \vdash_{div} n = 0)$ in G. Let x and y be two nodes of G such that $n = 1 \vdash_{div} n = 0$ in $G \wedge (x = y)$. Then $(x = 1 \vdash_{div} n = 0) \wedge (x = 0 \vdash_{div} n = 0)$ in $G \wedge (x = y)$.*

Proof. Let $G' = (G \wedge (x = y))$.

1. $\neg(n = 1 \vdash_{div} n = 0$ *in* $G)$ — Hypothesis
2. $n = 1 \vdash_{div} n = 0$ *in* G' — Hypothesis
3. $\neg(\exists p$ *partition of* G, $\exists fg$ *leaves of* p $/ f \neq g$, $ipath(n, f, G) \wedge ipath(n, g, G))$ — Proposition 5, 1
4. $(p$ *partition of* $G') \wedge (f$ *leaf of* $p) \wedge (g$ *leaf of* $p) \wedge f \neq g \wedge ipath(n, f, G') \wedge ipath(n, g, G')$ — Proposition 5, 2

If x is neither in an inclusion path $n - f$ nor $n - g$ in G' :

5. $\neg(ipath(n, x, G') \wedge ipath(x, f, G'))$ — Hypothesis
6. $\neg(ipath(n, x, G') \wedge ipath(x, g, G'))$ — Hypothesis

$ipath(n, f, G)$ — Proposition 8, 4, 5

$ipath(n, g, G)$ — Proposition 8, 4, 6

Contradiction with 3.

otherwise, x is in an inclusion path $n - f$ or $n - g$ in G'

$((ipath(n, x, G') \wedge ipath(x, f, G')) \vee (ipath(n, x, G') \wedge ipath(x, g, G')))$ — Hypothesis

By symmetry of variables f and g, only one case is necessary to examine :

7. $ipath(n, x, G') \wedge ipath(x, f, G')$ — Hypothesis

By Propositions 4 and 5 and facts 7 and 4 in G' we have :

$x = 0 \vdash_{div} n = 0$

$x = 1 \vdash_{div} f = 1 \vdash_{div} g = 0 \vdash_{div} n = 0$ □

Proposition 10. *Let G be a graph. Let n be a node of G such that $\neg(n = 1 \vdash_{div} n = 0$ in $G)$. Let p be a partition not in G such that $n = 1 \vdash_{div} n = 0$ in $G \wedge p$. Then $(\exists f$ leaf of p / $(f = 1 \vdash_{div} n = 0) \wedge (f = 0 \vdash_{div} n = 0)$ in $G \wedge p)$.*

Proof. Let p be a partition not in G :

1. $\neg(n = 1 \vdash_{div} n = 0$ *in* $G)$ — Hypothesis
2. $n = 1 \vdash_{div} n = 0$ *in* $(G \wedge p)$ — Hypothesis
3. $(q$ *partition of* $(G \wedge p)) \wedge (f$ *leaf of* $q) \wedge (g$ *leaf of* $q) \wedge f \neq g \wedge ipath(n, f, G \wedge p) \wedge ipath(n, g, G \wedge p)$ — Prop. 5, 2

If $p \neq q$, then q is a partition of G.

By Props. 4 and 5 and fact 3 :

$n = 1 \vdash_{div} n = 0$ *in* G

Contradiction with 1.

otherwise, $p = q$:

4. $(f\ leaf\ of\ p)\ \wedge\ (g\ leaf\ of\ p)\ \wedge\ f \neq g$
   $\wedge\ ipath(n, f, G \wedge p) \wedge ipath(n, g, G \wedge p)$ Subst. p to q in 3

By Props. 4 and 5 and fact 4, in $G \wedge p$ we have :
$f = 0 \vdash_{div} n = 0$
$f = 1 \vdash_{div} g = 0 \vdash_{div} n = 0$ □

For a graph with n nodes, the divergent propagation is in $O(n)$. Testing every node, or only the leaves of the graph, with the divergent propagation is in $O(n^2)$. If m is the maximum number of nodes in a partition, the bi-propagation is in $O(nm)$. For graphs of bipartitions, $m = 3$ and the bi-propagation is in $O(n)$.

4.1 Incremental Construction of Graphs of Partitions

Proposition 9 does not suppose that $G \rightarrow (x = y)$. The proposition is valid even when the merging is imposed. Similarly, Proposition 10 does not suppose that $G \rightarrow p$. This means that adding partitions and merging nodes may be used as deduction or construction operations. On each modification, the graph is simplified. This spontaneous simplification mixes gaussian elimination and bi-propagation to suppress contradictory nodes and merge equivalent nodes.

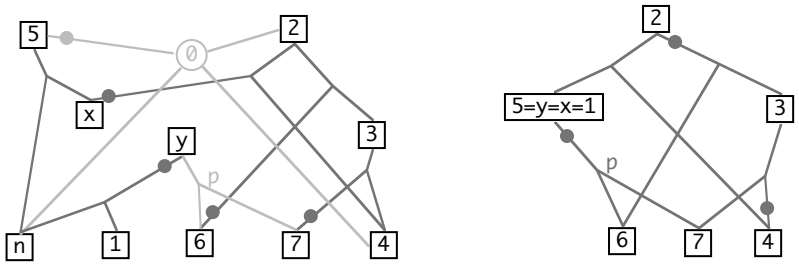


Fig. 2. The linear component of a graph. The equations derived by gaussian elimination which are not derivable from a single partition are shown as grey hyperlinks with a circled center. The defined variable of each equation is the dotted node.

Example 2. In Fig. 2, when partition p is added to the left graph, nodes x and y are merged by gaussian elimination, then node n is suppressed by bi-propagation, giving the right graph.

4.2 The Simplifier

The simplifier is a loop which effectively modifies the graph as required by equations $x = 0$, $x = y$ and pairs of equations $\{x = V, y = V\}$ of the solved linear

system. Each modification of the graph may add new equations to the system, either by gaussian elimination or by bi-propagation.

The necessity to physically suppress nodes and modify partitions makes this simplification difficult to implement. After each modification in a causal sequence, no pointer on the graph is safe. Finding safe pointers is a delicate question. Here, the modulo 2 linear reasoning is an ideal solution. At each equation $x = 0$ or $x = y$ deduced, the operation "add the equation to the system" only modifies the linear system, and the pointers on the graph are unaffected. When a node is effectively suppressed or a partition modified or suppressed, the linear equations of the solved system act as safe pointers on the graph.

5 Experimental Results

A system called GP has been implemented, including a graphical editor, the simplifier and some inference engines. GP is written in C++ under the MAC OS X + CARBON environment. All experiments were done on a 1.5ghz intel processor with 1GB of memory. Computation times are in seconds.

It is not clear how to compare a generalistic simplifier with a SAT solver. Satisfiability tests find individual solutions rather than restrict to what is deducible from the hypotheses. SAT solvers may however be used as subtools to make exact deductions. Using a SAT solver to test for contradiction on every node and for equivalence on every pair of nodes makes a complete generalistic boolean simplifier but is definitely not practical. So the comparisons in Tables 1 and 2 are simply meant to experimentally confirm the need to distinguish spontaneous simplifications from other goal oriented inferences. Two SAT solvers were tested, Zchaff [7] and Minisat 2.0 [2] on cnf translations of the valuated graphs.

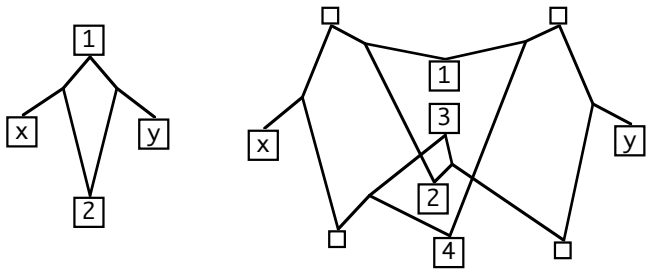


Fig. 3. Random graphs of depth one and two for the mcbin construction.

The first experiment uses random graphs of a very simple conception. Two binary trees of alternated bipartitions are randomly connected by their leaves. Figure 3 shows graphs of depth one and two. Nodes x and y are equivalent. The valuation $\{x = 0, y = 1\}$ makes the graphs contradictory. The mcbin7 file is a

Table 1. This experiment illustrates the difficulty for SAT solvers to cope with the linear component which is immediately available from graphs of partitions. All instances are unsatisfiable.

	#vars	#clauses	zchaff	Minisat2	GP
mcbin6	190	506	91.17	14.2	0.00
mcbin7	382	1018	>20000	>20000	0.00
mcbin8	766	2042	-	-	0.01
mcbin9	1534	4090	-	-	0.02
mcbin10	3070	8186	-	-	0.06
mcbin11	6142	16378	-	-	0.19

graph of depth 7 and this is where SAT solvers based on the DPLL procedure fail. The contradiction is derivable by gaussian elimination from the linear system derivable from the graph. The GP simplifier will also merge x and y if there is no valuation.

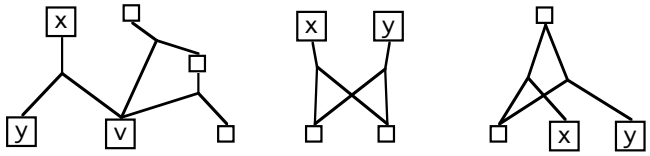


Fig. 4. Three simple graphs which deduce $x = y$.

Table 2. Computation times on random graphs involving a great number of node suppressions and mergings. All instances are satisfiable.

	#vars	#clauses	Minisat2	gaussian elim	simplif
s500	1504	4008	0.02	0.01	0.16
s1000	3004	8008	0.05	0.01	0.40
s1500	4504	12008	0.07	0.02	0.66
s2000	6004	16008	0.10	0.02	1.03

In the second experiment, graphs are randomly constructed by repeatedly replacing a single node x by one of the three graphs of Fig. 4. On simplification, nodes x and y will be merged back into x. The "gaussian elim" column of Table 2 is the time of the first gaussian elimination only. The computation times show that despite its cubic complexity, gaussian elimination is a fast process. The

"simplif" column is the remaining time of the simplifier, including the simplifications of the linear system (using a sub-operation of gaussian elimination) every time two nodes are merged. In comparison with Minisat2 and gaussian elimination, the reconstruction operations of the GP simplifier are the most expensive part. What justifies this is that reconstruction operations are not performed by the preprocessor of Minisat2, based on Variable Elimination Resolution, which seeks for a satisfiable assignment. The GP simplifications are deductions, not inferences. The simplified graph is still equivalent to the original graph and contains all its non contradictory variables, while Variable Elimination Resolution only preserves satisfiability.

6 Future Work

The actual simplifier operates on non valuated graphs. No deduction is done that is useful only in the context of a valuation. Such deductions can anyway end up in globally useful simplifications. The sequence of decisions in backtrack search is an obvious illustration.

The next step is to augment the convergent propagation into a more powerful deduction mechanism able to extend the current valuation of a graph. One problem to overcome is the great quantity of bipartitions added if each unit propagation is memorized and the newly introduced node is classified in the graph by convergent propagation. Solutions to this problem actually appear to require some specialized linear reasonings beside gaussian elimination.

7 Conclusion

This paper has presented the graphs of partitions. This logical formalism may represent arbitrary boolean constraints. Its originality is that modulo 2 linear equations are immediately available from each partition. Whenever a new partition is deduced, a linear equation is deduced and gaussian elimination, mixed with a process called bi-propagation, simplifies the graph. Two propositions restrict the number of nodes to test for contradiction by bi-propagation. Experiments show that the simplifier is rapid enough to be used as an interactive tool for the bookkeeping of a data base of constraints.

Spontaneous simplification has been distinguished from preprocessing and solving problem instances. It is not actually possible to *demonstrate* the necessity of such mechanisms. Such a demonstration would consist in a reformulation algorithm which would adapt to any formulation of a problem and provide a representation that makes it tractable. Common intuition assumes the usefulness of finding contradictory or equivalent expressions. This is an intermediate step towards more ambitious automatic reencoding mechanisms.

References

1. P. Baumgartner and F. Massacci. The taming of the (x)or. In *Computational Logic – CL 2000*, volume 1861, pages 508–522. Springer, 2000.

2. Niklas Een and Niklas Sörensson. An extensible sat-solver [ver 1.2]. In *SAT 2003*, pages 502–508, 2003.
3. D. Goossens. Bipartitions et equations lineaires. In *Programmation en logique avec contraintes (JFPLC 2003)*, pages 299–302, Amiens, France, June 2003. Hermes.
4. M. Heule and H. van Maaren. Aligning cnf- and equivalence-reasoning. In *Seventh International Conference on Theory and Applications of Satisfiability Testing (SAT 2004)*, Vancouver, 2004.
5. I. Lynce and J. Marques-Silva. The puzzling role of simplification in propositional satisfiability. In *EPIA'01 Workshop on Constraint Satisfaction and Operational Research Techniques for Problem Solving (EPIA-CSOR)*, 2001.
6. Li Chu Min. Integrating Equivalency Reasoning into Davis-Putnam Procedure. In *Proceedings of AAAI-2000*, Austin, Texas, USA, 2000.
7. Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, 2001.
8. G. S. Huang N. Dershowitz, J. Hsiang and D. Kaiss. Boolean ring satisfiability. In *Seventh International Conference on Theory and Applications of Satisfiability Testing (SAT 2004)*, pages 281–286, Vancouver, 2004.
9. M.A. Papalaskaris and L.K. Schubert. Parts Inference: Closed and Semi-closed Partitioning graphs. In *Proceedings of the 7th IJCAI*, pages 304–309, Vancouver, 1981.
10. T.J. Schaefer. The Complexity of Satisfiability Problems. In *Proc. 10th Ann ACM Symp. on Theory of Computing*, pages 216–226, New York, 1978. Assoc. for Comput. Mach.
11. S. Subbarayan and Dhiraj Pradhan. Niver: Non increasing variable elimination resolution for preprocessing sat instances. *In Selected Revised Papers of International Conference on Theory and Applications of Satisfiability Testing Conference (Springer LNCS)*, pages 276–291, May 2004.
12. J.P. Warners and H. van Maaren. A two phase algorithm for solving a class of hard satisfiability problems. *Operations Research Letters*, 23:81–88, 1998.
13. B. Zanuttini. Approximating propositional knowledge with affine formulas. In *Proc. 15th European Conference on Artificial Intelligence (ECAI'02)*, pages 287–291. IOS Press, 2002.

Generating Implied Boolean Constraints via Singleton Consistency

Roman Barták

Charles University in Prague, Faculty of Mathematics and Physics
Malostranské nám. 2/25, 118 00 Praha 1, Czech Republic
roman.bartak@mff.cuni.cz

Abstract. Though there exist some rules of thumb for design of good models for solving constraint satisfaction problems, the modeling process still belongs more to art than to science. Moreover, as new global constraints and search techniques are being developed, the modeling process is becoming even more complicated and a lot of effort and experience is required from the user. Hence (semi-) automated tools for improving efficiency of constraint models are highly desirable. The paper presents a low-information technique for discovering implied Boolean constraints in the form of equivalences, exclusions, and dependencies for any constraint model with (some) Boolean variables. The technique is not only completely independent of the constraint model (therefore a low-information technique), but it is also easy to implement because it is based on ideas of singleton consistency. Despite its simplicity, the proposed technique proved itself to be surprisingly efficient in our experiments.

Keywords: implied constraints, reformulation, singleton consistency, SAT.

1 Introduction

Problem formulation is critical for efficient problem solving in formalisms like SAT (satisfiability testing), LP (linear programming), or CS (constraint satisfaction). LP and SAT formalisms are quite restricted, to linear inequalities in LP and logical formulas in SAT. Hence problem formulation is studied for a long time in LP and SAT because it is not always easy to express real-life constraints using linear inequalities or logical formulas. We cay say that the problem formulation is the core of courses for normal users of LP and SAT, while the solving techniques are studied primarily by experts and researchers contributing to improving the solving techniques. Opposite to SAT and LP, the CS formalism is very rich concerning its expressivity (any constraint can be directly modeled there). Hence the users get a big freedom in expressing their problems as constraint satisfaction problems which has some negative consequences. First, because the solvers need to cover the generality of the problem formulation, it is hard to improve their efficiency, and, actually, we have not observed the dramatic increase of speed of constraint solvers similar to SAT and LP solvers. Second, the main burden on efficient problem solving is on the user who must understand the details of the solving process to formulate the problem in an

efficient way. Note that sometimes a small change in the model, such as adding a single constraint, may dramatically influence efficiency of problem solving which makes the modeling task even more complicated. There exist some rules of thumb how to design efficient constraint models [10,13], but constraint modeling is still assumed to be more art than science. There exist some automated techniques for on-fly problem re-formulation such as detecting and breaking symmetries during search (for a short survey see [13]) or no-good recording (introduced in [14] and formally described in [6]). Usually the problem (re-)formulation is up to the user by using techniques such as adding symmetry breaking or implied constraints, encoding parts of the problem using specialized global constraints, or adding dominance rules.

In this paper, we address the problem of fully automated generation of useful implied constraints in constraint satisfaction problems. Informally speaking, by a useful implied constraint we mean a constraint than is deduced from the existing model (hence implied) and that positively contributes to faster problem solving (hence useful). A fully automated technique means that the implied constraints are generated for any given constraint model without any user intervention. According to the principle that the best constraint model will be the one in which information is propagated first [10] we are trying to generate implied constraints that propagate more than the existing constraints (remove more inconsistencies from the model). Recall that more inconsistencies can be easily removed from any constraint model by applying a stronger consistency technique, for example by using path consistency instead of arc consistency. However, the main problem with stronger consistency techniques is their time and space complexity which disqualifies these techniques from being used repeatedly in the nodes of the search tree. Naturally, stronger consistency techniques can be applied once before the search starts but then their effect is limited to removing initially inconsistent values from variables' domains. We propose to exploit information from these stronger consistency techniques in the form of implied constraints that are deduced during the initial consistency process and added to the constraint model. In particular, we propose to use singleton arc consistency [5] to deduce new constraints between Boolean variables in the problem. The rationale for using singleton arc consistency (SAC) is that this meta-technique is easy to implement on top of any constraint model (singleton consistency is a meta-technique because it works on top of other "plain" consistency techniques such as arc consistency or path consistency). The reasons for restricting to Boolean variables are twofold. First, singleton consistency is an expensive technique especially when applied to variables with large domains so Boolean variables seem to be a good compromise. Second, we need to specify the particular form of constraints that we are learning, which is easier for Boolean variables. To be more specific, at this stage we are learning only the equivalence, implication, and exclusion constraints. In [1] we already showed that SAC over Boolean variables contributes a lot to removing initial inconsistencies so our hope is that the constraints derived from SAC can further help in problem solving.

The paper is organized as follows. After giving the initial motivation for our work, we will define more formally the used notions and techniques. Then we will present the core of the proposed technique and the paper will be concluded by an experimental section showing the benefits and detriments of the proposed method.

For now, we can reveal that despite the simplicity of the proposed method, the experiments showed surprising speed-ups for some problems.

2 Motivation

In [2] we proposed a novel constraint model for description of temporal networks with alternative routes similar to [4]. Briefly speaking, this model consists of a directed acyclic graph or in general a Simple Temporal Network [7], where the nodes are annotated by Boolean validity variables. There are special constraints between the validity variables describing logical relations between the nodes (we call them parallel and alternative branching). These constraints specify which nodes should be selected together to form one of the possible alternative routes through the network. Figure 1 shows an example of alternative branching together with a constraint model describing the relations between the validity variables

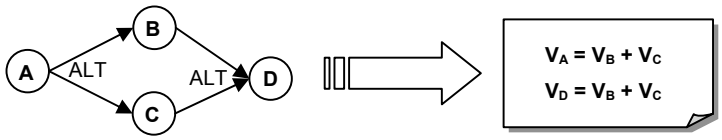


Fig. 1. A simple graph with alternative branching (left) and its formulation as a constraint satisfaction problem (left) over the validity variables.

The above model is useful for description of manufacturing scheduling problems, but it suffers from several drawbacks. The main issue is that the problem of deciding which nodes are valid in the network is NP-complete in general [2]. Hence, opposite to Simple Temporal Networks [7] we cannot expect a complete polynomial constraint propagation technique that removes all inconsistencies from the constraint model. For example, the constraint model in Figure 1 cannot discover, using standard (generalized) arc consistency, that $V_A = 1$ if V_D is set to 1 (and vice versa). In [3] we proposed some pre-processing rules that can deduce implied constraints improving the filtering power of the constraint model. In particular, we focused on discovering (some) equivalent nodes, that is, the nodes whose validity status is identical in all feasible solutions (such as nodes A and D in Figure 1). Unfortunately, we also showed there that the problem whether two nodes are equivalent is also NP-hard. Our pre-processing rules from [3] are based on contracting the graph describing the problem and it is not easy to implement them and to extend them to other problems. Moreover, this method is looking only for equivalent nodes and ignores other useful relations such as dependencies and exclusions.

The above problem is not the only problem combining Boolean and temporal variables. Fages [8] studies a constraint model for describing and solving min-cutset problems and log-based reconciliation problems. Again, there are Boolean validity variables, which can be connected by dependency constraints in case of log-based reconciliation problems, and ordering variables describing the order of the nodes in a linear sequence of nodes (to model acyclicity of the selected sub-graph). We believe

that there are many other real-life problems where Boolean variables are combined with numerical variables. Our learning method might be useful for such problems to discover implied constraints between the Boolean variables that also take in account the other constraints. Naturally, we can learn implied constraints in problems with Boolean variables only, such as SAT problems.

3 Preliminaries

A *constraint satisfaction problem* (CSP) P is a triple (X, D, C), where X is a finite set of decision variables, for each $x_i \in X$, $D_i \in D$ is a finite set of possible values for the variable x_i (the domain), and C is a finite set of constraints. A constraint is a relation over a subset of variables that restricts possible combinations of values to be assigned to the variables. Formally, a constraint is a subset of the Cartesian product of the domains of the constrained variables. We call the variable Boolean if its domain consists of two values {0, 1} (or similarly {*false*, *true*}). A *solution to a CSP* is a complete assignment of values to the variables such that the values are taken from respective domains and all the constraints are satisfied. We say that a constraint C is (generalized) *arc consistent* if for any value in the domain of any constrained variable, there exist values in the domains of the remaining constrained variables in such a way that the value tuple satisfies the constraint. This value tuple is called a support for the value. Note that the notion arc consistency is usually used for binary constraints only, while generalized arc consistency is used for n-ary constraints. For simplicity reasons we will use the term arc consistency independently of constraint's arity. The CSP is *arc consistent* (AC) if all the constraints are arc consistent and no domain is empty. To make the problem arc consistent, it is enough to remove values that have no support (in some constraint) until only values with a support (in each constraint) remain in the domains. If any domain becomes empty then the problem has no solution. We say that a value a in the domain of some variable x_i is *singleton arc consistent* if the problem $P|x_i=a$ can be made arc consistent, where $P|x_i=a$ is a CSP derived from P by reducing the domain of variable x_i to $\{a\}$. The CSP is *singleton arc consistent* (SAC) if all values in variables' domains are singleton arc consistent. Again, the problem can be made SAC by removing all SAC inconsistent values from the domains. Figure 2 shows an example of a CSP and its AC and SAC forms.

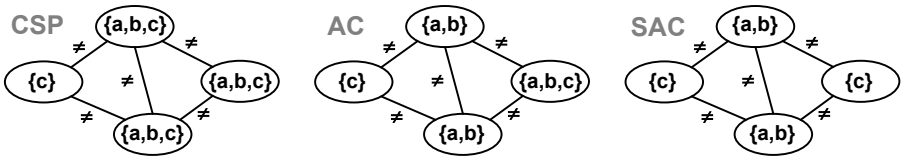


Fig. 2. A graph representation of a CSP, an arc consistent problem, and a singleton arc consistent problem (from left to right).

Assume now the constraint satisfaction problem with Boolean variables A, B, C, and D and with constraints A = B + C and D = B + C (like in Figure 1). This problem is both AC and SAC. Now assume that we assign value 1 to variable A. The problem remains AC but it is not SAC because value 0 cannot be assigned to variable D. This is an example of weak domain pruning in our temporal networks with alternatives. If we now include constraint A = D and make the extended problem AC then value 0 is removed from the domain of D by AC. Clearly, any assignment satisfying the original constraints also satisfies this added constraint. Hence we call this constraint *implied*, because the constraint is logically implied by the original constraints (sometimes, these constraints are also called *redundant*). Our goal is to find such implied constraints that contribute to stronger domain filtering.

4 Learning via SAC

As we already mentioned in the introduction and motivation, our original research goal was to easily identify some equivalent nodes in the temporal networks with alternatives. Recall, that finding all equivalent nodes is an NP-hard problem [3] so we focused only on finding equivalences similar to those presented in Figure 1 (nodes A and D). An easy way, how to identify such equivalences, is a trial-and-error method similar to shallow backtracking or SAC. Basically, we will try to assign values to pairs of variables and if we find that only identical values can be assigned to the variables then we deduce that the variables are equivalent (they must be assigned to the same value in any solution). As a side effect, we can also discover some dependencies between the variables (if 1 is assigned to B then 1 must be assigned to A) and exclusions between the variables (either B or C must be assigned to 0 or in other words it is not possible to assign 1 to both variables B and C).

We will now present the learning method for an arbitrary constraint satisfaction problem P. Recall, that we will only learn specific logical relations between the Boolean variables of P. We will gradually try to assign values to variables and each time we try the assignment, this assignment is propagated to other variables (the problem is made AC). If the assignment leads to a failure then we know that the other value in the domain must be assigned to the variable (recall that we are working with Boolean variables). The whole learning process consists of two stages.

First, we collect information about which variables are instantiated after assigning value 1 to some variable A. We distinguish between *directly instantiated variables*, that is, those variables that are instantiated by making the problem $P|_{A=1}$ arc consistent (one value in the variable domain is refuted by AC so the other value is used), and *indirectly instantiated variables*, that is, those variables where we found their value by refuting the other value in a SAC-like style (AC did not prune the domain, but when we try to assign a particular value to the variable it leads to a failure so the other value is used). Informally speaking, if we assign value 1 to variable A and make the problem arc consistent then all variables that are newly instantiated are directly instantiated variables. Indirectly instantiated variables are those variables B that are not instantiated by AC in $P|_{A=1}$ but for which only one value is compatible with A = 1 because if the other value is assigned to B, it leads to a failure after making the

problem AC (see procedure `Learn` below). More formally, let B be a non-instantiated (free) Boolean variable in $AC(P|_{A=1})$, where AC(P) is the arc consistent form of problem P (inconsistent values are removed from the domains of variables). If $P|_{A=1,B=0}$ is not arc consistent then value 0 cannot be assigned to B, hence value 1 must be used for B. Symmetrically, we can deduce that value 0 must be assigned to B if $P|_{A=1,B=1}$ is not arc consistent. Together, we can deduce which value must be used for B if value 1 is assigned to A. If both values for B are feasible then no information is deduced. If no value for B is feasible then value 1 cannot be used for A and hence A must be instantiated to 0. Note that information about indirectly instantiated variables is very important because it will help us to deduce implied constraints that improve propagation of the original constraint model. More formally, we are looking for implied constraints C such that $AC(P|_C) \subset AC(P)$, where $P|_C$ is a problem P with added constraint C and the subset relation means that all domains in $AC(P|_C)$ are subsets of relevant domains in AC(P) and at least one domain in $AC(P|_C)$ is a strict subset of the relevant domain in AC(P). In other words, constraint C helps in removing more inconsistencies from problem P.

The learning stage deduces three types of implied constraints. If $B = 0$ is indirectly deduced from the assignment $A = 1$ and $A = 0$ is indirectly deduced from the assignment $B = 1$ then the pair {A, B} forms an exclusion, which is an implied *exclusion constraint* ($A = 0 \vee B = 0$). Notice that this constraint really improves propagation because for example if 1 is assigned to A then the constraint immediately deduces $B = 0$, while the original set of constraints deduced no pruning for B. Similarly, if $B = 1$ is indirectly deduced from the assignment $A = 1$ then B depends on A, which is an implied *dependency constraint* ($A = 1 \Rightarrow B = 1$). Again, this constraint improves propagation. Note that we introduce this constraint only if variables A and B are not found to be equivalent. The equivalent variables are found using the following procedure. We construct a directed acyclic graph where the nodes correspond to the variables and the arcs correspond to the dependencies between the variables. These dependencies are found in the first stage, we assume both direct dependencies discovered by the AC propagation and indirect dependencies discovered by the SAC-like propagation. Strongly connected components of this graph form *equivalence classes of variables*. Note that if A and B are in a strongly connected component then ($A = 1 \Rightarrow^* B = 1$) and ($B = 1 \Rightarrow^* A = 1$), where $\Rightarrow^*$ is a transitive closure of relation $\Rightarrow$. All equivalent variables must be assigned to the same value in any solution so we can put equality constraint between these variables.

The following code of procedure `Learn` shows both the data collecting stage and the learning stage of our method. BoolVars(P) is a set of not-yet instantiated Boolean variables in P, doms(P) are domains of P, $D_X = \{V\}$ means that the domain of variable X consists of one element V, and AC(P) is the arc consistent form of problem P (AC(P) = fail if problem P cannot be made arc consistent).

The main advantage of the proposed method is simplicity and generality. Thanks to meta-nature of singleton consistency it can be implemented easily in any constraint solver and it works with any constraint satisfaction problem (even if global constraints and non-Boolean variables are included). The time complexity of the data collection stage is $O(n^2.|AC|)$, where n is the number of Boolean variables and |AC| is the complexity to make the problem arc consistent. Strongly connected components of the dependency graph can be found in time not greater than $O(n^2)$ and exclusions

and dependencies are generated in time $O(n^2)$. Clearly, majority of time to learn implied constraints by the above method is spent by collection information using the SAC-like method.

```
procedure Learn (P: CSP)
  for each A in BoolVars(P) do // data collecting stage
    Q ← AC(P|A=1)
    Direct(A) ← { X/V | DX = {V} in doms(Q)}
    for each B in BoolVars(Q) s.t. A ≠ B & Q ≠ fail do
      if AC(Q|B=0) = fail then
        Q ← AC(Q|B=1)
      else if AC(Q|B=1) = fail then
        Q ← AC(Q|B=0)
    end for
    Indirect(A) ← { X/V | DX = {V} in doms(Q)} – Direct(A)
    if Q = fail then
      P ← AC(P|A=0)
      if P = fail then stop with failure
  end for
  // learning stage
  G ← (BoolVars(P), {(A,B) | B/1 ∈ Direct(A) ∪ Indirect(A)})
  Equiv ← StronglyConnectedComponents(G)
  Excl ← { {A,B} | B/0 ∈ Indirect(A) & A/0 ∈ Indirect(B)}
  Deps ← { (A,B) | B/1 ∈ Indirect(A) & ¬ {A,B} ⊆ X ∈ Equiv}
  return (Equiv, Excl, Deps)
end Learn
```

5 Implementation and Experiments

To evaluate whether our learning technique is useful for problem solving we implemented the learning technique in SICStus Prolog 3.12.3 and tested it on 1.8 GHz Pentium 4 machine running under Windows XP. Note that we used a naïve (non-optimal) implementation of the SAC algorithm that is called SAC-1 [5]. This algorithm simply assigns a value to the variable and propagates this assignment via standard arc-consistency algorithm. The algorithm does not pass any data structures between several runs which makes it non-optimal. Nevertheless, its greatest advantage is that the implementation is very easy and can be realized in virtually any constraint solver. For the experiments we used existing benchmarks for min-cutset problems [11] and a dozen of benchmarks for SAT problems [9].

5.1 Learning for CSP

In our first experiment, we compared efficiency of the original constraint model for min-cutset problems from [8] with the same constraint model enhanced by the learned implied constraints. Note that these constraint models contain both Boolean variables

(validity) and integer variables (ordering of nodes). Recall that the min-cutset problem consists of finding the largest subset of nodes such that the sub-graph induced by these nodes does not contain a cycle. So it is an optimization problem. We used the data set from [11] with 50 activities and a variable number of precedence relations. Figure 3 shows the comparison of above models both in the runtime (milliseconds) and in the number of backtracks. It is important to say that the runtime for the enhanced model consists of the time to learn the implied constraints and the time to solve the problem to optimality (using the branch-and-bound method). The time to learn the implied constraints is negligible there (from 80 to 841 milliseconds) and hence we do not show that time separately in the graphs. We used the well-known Brélaz variable ordering heuristic also known as dom+deg heuristic (the variables with the smallest domain are instantiated first, ties broken by preferring the most constrained variables).

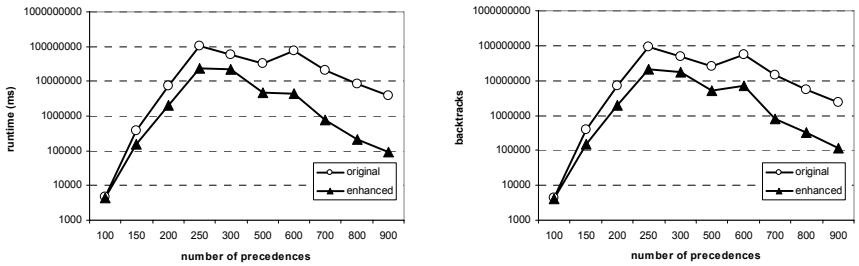


Fig. 3. Comparison of runtimes (milliseconds) and the number of backtracks for the original model of min-cutset problems and the model enhanced by the learned implied constraints with the Brélaz variable ordering heuristic.

The graphs in Figure 3 show a significant decrease of the runtime and of the number of backtracks, which is a promising result especially taking in account that the time to learn is included in the overall time. This decrease is mainly due to the learned exclusion constraints which capture cycles in the graph (one node in the exclusion must be invalid to make the graph acyclic). Clearly, the Brélaz heuristic is also influenced by adding constraints to the model so the implied constraints may change the ordering of variables during search and hence influence efficiency. As we want to see also the effect of implied constraints on pruning the search space, we need to use exactly the same search procedure for both models. The straightforward approach is to use a static variable ordering. Figure 4 shows the comparison of both models using the static variable ordering heuristic. Again, we used the branch-and-bound method to solve the problem to optimality. Due to time reasons, we used a cut-off limit 300 000 000 milliseconds (>83 hours) for a single run so the most complex problems (200 - 300, and 600 precedences) were not solved to optimality for the original model and hence information about the number of backtracks is missing.

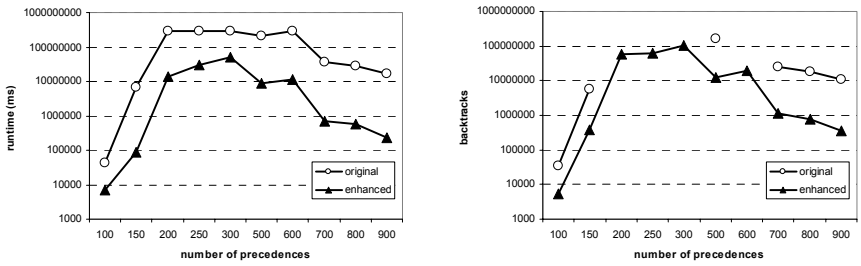


Fig. 4. Comparison of runtimes (milliseconds) and the number of backtracks for the original model of min-cutset problems and the model enhanced by the learned implied constraints with the static variable ordering heuristic (a logarithmic scale).

Again, the enhanced model beats the original model and shows a significant speedup. Moreover, by comparing both experiments, we can see that the learned constraints not only pruned more the search space by stronger domain filtering (which was our original goal) but in combination with the Brélaz heuristic they also make the search faster by focusing the search algorithm to critical (the most constrained) variables.

5.2 Learning for SAT Problems

Because our method works primarily with Boolean variables, the natural benchmark to test efficiency of the method was using SAT problems. We take several problem classes from [9], namely logistics problems from AI planning, all-interval problems, and quasigroup (Latin square) problems and encoded the problems in a straightforward way as CSPs. The choice of problem classes was driven by the idea that structured problems may lead to more and stronger implied constraints. It would be surely better to do more extensive tests with other problem classes, but a limited computation time forced us to select only few most promising classes. Again we used the Brélaz variable ordering heuristic in the search procedure which was backtracking search with maintaining arc consistency. Table 1 summarizes the results, it shows the problem size (the number of Boolean variables), the number of backtracks and the time to solve the problems (for the enhanced model the time includes both the time to learn as well as the time solve the problem), and the time for learning.

Table 1. Comparison of solving efficiency of the original and enhanced constraint models for selected SAT problems (the smallest #backtracks / runtime is in bold).

instance	size	original		enhanced		
		backtracks	runtime (ms)	backtracks	overall time (ms)	time to learn (ms)
logistics.a	828	>159827502	>60000000	**4**	**53677**	53657
logistics.b	843	>107546059	>60000000	**38494**	**91622**	65955
logistics.c	1141	>95990563	>60000000	**26195**	**165537**	150776
logistics.d	4713	>38809049	>60000000	**5738102**	**28866167**	16116604
ais6	61	16	**10**	**3**	400	390
ais8	113	**178**	**120**	523	3435	3155
ais10	181	3008	**2914**	**118**	14911	14811
ais12	265	66119	80386	**140**	**49091**	48921
qg1-07	343	26	**811**	0	146371	146291
qg1-08	512	**331474**	**12445947**	1791551	59608683	886605
qg2-07	343	34	**1061**	**0**	178987	178906
qg2-08	512	213992	8005862	213992	**7980054**	1053394
qg3-08	512	26	**170**	**22**	68018	67908
qg3-09	729	357521	2216758	**25246**	**343845**	233917
qg4-08	512	2956	**12839**	**367**	68088	66556
qg4-09	729	614	**3925**	**86**	225324	224934
qg5-09	729	1525	22573	**0**	***1933***	*1933*
qg5-10	1000	119894	2647697	**0**	***61318***	*61318*
qg5-11	1331	>1741008	>60000000	**0**	**855000**	854880
qg5-12	1728	>1195753	>60000000	**0**	***6467810***	*6467810*
qg5-13	2197	>802393	>60000000	**41641**	**23622817**	21695532
qg6-09	729	177	**2304**	**0**	51143	51113
qg6-10	1000	12234	238493	**0**	***63732***	*63732*
qg6-11	1331	1668478	34617658	**4545**	**3233200**	3153716
qg6-12	1728	>2512643	>60000000	**586472**	**22669216**	7159264
qg7-09	729	0	**40**	0	53337	53297
qg7-10	1000	348	**6930**	**0**	*46557*	*46557*
qg7-11	1331	27777	674701	**0**	***429658***	*429658*
qg7-12	1728	>2239230	>60000000	**148648**	**10344354**	6560683
qg7-13	2197	**261**	**14101**	525428	31893597	13893597

The experimental results show some interesting features of the method. First, the model enhanced by the learned implied constraints was frequently solved faster and using a smaller number of backtracks than the original model. The smaller number of backtracks is not that surprising, because the implied constraints contribute to pruning the search space. However, a shorter overall runtime for the enhanced model is a nice result, especially taking in account that the overall runtime includes the time to learn the implied constraints. The speed-up is especially interesting in the logistics problems, where the learning method deduced many exclusion constraints (probably thanks to the nature of the problem) which contributed a lot to decreasing the search space. The few examples when solving required more backtracks for the enhanced model (ais8, qg1-08, and qg7-13) can be explained by "confusing" the variable ordering heuristic by the implied constraints. Figure 3 and 4 showed that adding implied constraints influenced significantly the Brélaz variable ordering heuristic which is clear – the labeled variables have Boolean domains so the not-yet instantiated variables are ordered primarily by using the number of constraints in

which they are involved. It may happen that in some problems this may lead to a wrong decision as no heuristic is perfect for all problems. It will be interesting to study further how the added implied constraints influence structure-guided variable ordering heuristics.

A second interesting feature is that for several quasigroup problems which have no feasible solution, the learning method proved infeasibility (in italics in Table 1) so no subsequent search was necessary to solve the problem. Again in most problems it was still faster than using the original constraint model. Finally, though we almost always improved the solving time, the overhead added by the learning method (the additional time to learn) was not negligible and the total time to solve the problem was sometimes worse than using the original model. This is especially visible in simple problems, where we spent a lot of time by learning, while in the meantime the backtracking search found easily the solution in the original model. This leads to a straightforward conclusion that if the original constraint model is easy to solve, it is useless to spent time by improving the model, for example by adding the implied constraints. Of course, the open question is how to find if the model is easy to solve.

5.3 Reformulation for SAT Solvers

In the previous section, we used SAT problems to demonstrate how the proposed learning method improves the solving time for these problems. However, we modeled the SAT problems using constraints and we used constraint satisfaction techniques to solve such models (combination of backtrack search and constraint propagation), which is surely not the best way to solve SAT problems. In the era of very fast SAT solvers, it might be interesting to find out if the implied constraints, that we learned using a constraint model, can also improve efficiency of the SAT solvers. We used one of the winning solvers in the SAT-RACE 2006 competition, RSat [12], to validate our hypothesis, that the learned constraints may also improve efficiency of SAT solvers. Table 2 shows the comparison of the number of backtracks, the number of decision (choice) points, and runtime for the original SAT problem and for the SAT problem with the added implied constraints. Again, we used the problem classes from [9].

There is clear evidence that the implied constraints decrease significantly the number of choice points of the RSat solver (and in most cases also the number of backtracks). This is an interesting result, because the RSat solver is using different solving techniques than the CSP solvers, to which our learning algorithm is targeted. Nevertheless, regarding the runtime the situation is different. Thought the difference is not big, the model enhanced by the implied constraints is slower in most cases. This may be explained by the additional overhead for processing a larger number of clauses. Note that for some models, the percent of the implied constraints is 20-30% of the original number of constraints so if the solver is fast, this increase of the model size will surely influence the runtime. Still, it is interesting to see that the learned implied constraints are generally useful to prune the search space and perhaps, for more complicated problems, their detection may pay-off even if we assume time to learn these constraints (Table 1).

Table 2. Comparison of solving efficiency of the original model and the model with learned constraints solved by RSAT solver (the smallest #backtracks / #decisions / runtime is in bold).

instance	original			enhanced		
	backtracks	decisions	runtime (ms)	backtracks	decisions	runtime (ms)
logistics.a	137	1394	**40**	**31**	**176**	50
logistics.b	251	2019	**60**	**119**	**558**	90
logistics.c	238	2999	**75**	**126**	**617**	80
logistics.d	**33**	547	**130**	42	**377**	1022
ais6	14	46	5	**0**	**11**	**0**
ais8	20	74	10	**0**	**22**	10
ais10	1142	1877	90	**0**	**37**	**20**
ais12	19	152	**25**	**0**	**56**	30
qg1-07	105	134	140	**44**	**72**	**130**
qg1-08	**4732**	**5608**	**1542**	18288	20528	8142
qg2-07	**35**	54	130	37	**53**	130
qg2-08	**14017**	**16270**	**6228**	45678	52308	31320
qg3-08	122	175	**40**	122	**153**	50
qg3-09	57294	65736	26137	**38434**	**44384**	**19027**
qg4-08	638	737	**100**	**586**	**667**	110
qg4-09	8	30	60	**6**	**23**	60
qg5-11	44	78	**230**	**0**	**4**	370
qg5-13	38617	48396	**36111**	**32971**	**38733**	39046
qg6-09	0	15	**70**	0	**3**	130
qg6-12	12386	14426	**7731**	**11171**	**13230**	7761
qg7-09	1	7	**70**	**0**	**3**	130
qg7-12	4052	5042	**1862**	**3360**	**4104**	1912
qg7-13	2716	4139	1592	**1375**	**1935**	**1131**

5.4 Learning Efficiency

The critical feature of the proposed method is efficiency of learning, that is, how much time we need to learn the implied constraints. In our current implementation, this time is given be the repeated calls to the SAC algorithm so the time depends a lot on the number of involved Boolean variables and also on the complexity of propagation (the number of constraints). The following figure shows the time for learning as a function of the number of involved Boolean variables for experiments from the previous sections (plus some additional SAT problems).

Clearly, due to the complexity of SAC, the proposed method is not appropriate for problems with a large number of Boolean variables. Based on our experiments, as a rough guideline, we can say that the method is reasonably applicable to problems with less than a thousand of Boolean variables. This seems small for SAT problems, but we believe it is a reasonable number of Boolean variables in CSP problems where non-Boolean variables are also included.

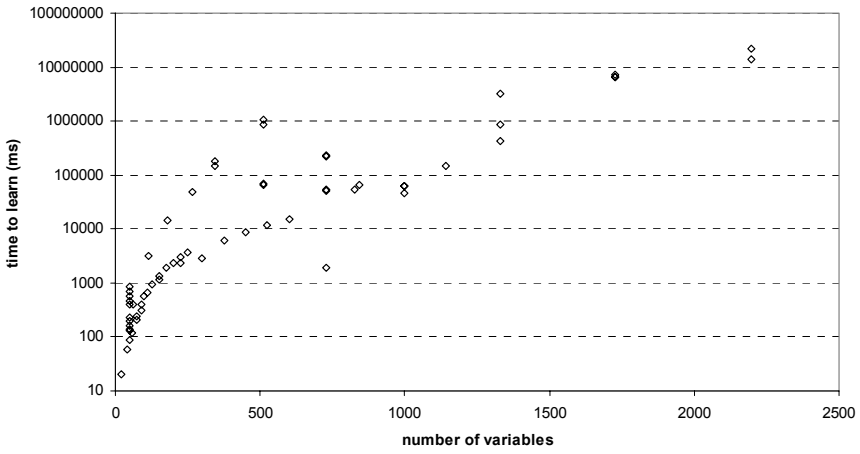


Fig. 5. Time to learn (in milliseconds) as a function of the number of involved Boolean variables (a logarithmic scale).

4 Conclusions

In the paper we proposed an easy to implement method for learning implied constraints over the Boolean variables in constraint satisfaction problems and we presented some preliminary experiments showing a surprisingly good behavior of this method. In the experiments we used naïve hand-crafted constraint models, that is, the models that a "standard" user would use to describe the problem as a CSP, so the nice speed-up is probably partly thanks to weak propagation in these models. Nevertheless, recall the holly grail of constraint processing – the user states the constraints and the solver provides a solution. For most users, it is natural to use the simplest constraint model to describe their problem and we showed that for such models, we can improve the speed of problem of solving.

To summarize the main advantages of the proposed method: it is easy to implement, it is independent of the input constraint model, and it contributes to speed-up of problem solving. The experiments also showed the significant drawback of the method – a long time to learn (an expected feature due to using SAC techniques). Clearly, the method is not appropriate for easy-to-solve problems where the time to learn is much larger than the time to solve the original constraint model. On the other hand, we did the majority of experiments with the SAT problems where all variables are Boolean, while the method is targeted to problem where only a fraction of variables is Boolean, such as the min-cutset problem. We believe that the method is appropriate to learn implied constraints for the base constraint model which is then extended by additional constraints to define a particular problem instance. So learning is done just once while solving is repeated many times. Then the time to learn is amortized by the repeated attempts to solve the problem. The time to learn can also be

decreased by identifying the pairs of variables that could be logically dependent. This may decrease the number of SAC checks. We did some preliminary experiments with the SAT problems, where we tried to check only the pairs of variables that are not "far each from other", but the results were disappointing – the system learned fewer implied constraints. Still, restricting the number of checked pairs of variables may be useful for some particular problems.

Note finally that the ideas presented in this paper for learning Boolean constraints using SAC can be extended to learning other type of constraints using other consistency techniques. However, as our experiments showed, it is necessary to find a trade-off between the time complexity and the benefit of learning.

Acknowledgments. The research is supported by the Czech Science Foundation under the contract no. 201/07/0205.

References

1. Barták, R.: A Flexible Constraint Model for Validating Plans with Durative Actions. In Planning, Scheduling and Constraint Satisfaction: From Theory to Practice. Frontiers in Artificial Intelligence and Applications, Vol. 117, IOS Press (2005), 39–48
2. Barták, R.; Čepek, O.: Temporal Networks with Alternatives: Complexity and Model. In Proceedings of the Twentieth International Florida AI Research Society Conference (FLAIRS 2007). AAAI Press (2007)
3. Barták, R.; Čepek, O.; Surynek, P.: Modelling Alternatives in Temporal Networks. In Proceedings of the 2007 IEEE Symposium on Computational Intelligence in Scheduling (CI-Sched 2007), IEEE Press (2007), 129–136
4. Beck, J.Ch.; Fox, M.S.: Scheduling Alternative Activities. In Proceedings of the National Conference on Artificial Intelligence, AAAI Press (1999), 680–687
5. Debruyne, R.; Bessière; C.: Some Practicable Filtering Techniques for the Constraint Satisfaction Problem. In Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI), Morgan Kaufmann (1997), 412–417
6. Dechter, R.: Learning while searching in constraint satisfaction problems. In Proceedings of the Fifth National Conference on Artificial Intelligence. AAAI Press (1986), 178–183
7. Dechter, R.; Meiri, I. and Pearl, J.: Temporal Constraint Networks. Artificial Intelligence 49 (1991) 61–95
8. Fages, F.: CLP versus LS on log-based reconciliation problems for nomadic applications. In Proceedings of ERCIM/CompulogNet Workshop on Constraints, Praha (2001)
9. Hoos, H.H.; Stützle, T.: SATLIB: An Online Resource for Research on SAT. In SAT 2000, IOS Press (2000) 283–292. SATLIB is available online at *www.satlib.org*.
10. Mariot, K.; Stuckey, P.J.: Programming with Constraints: An Introduction. The MIT Press, (1998)
11. Pardalos, P.M.; Qian, T.; Resende, M.G.: A greedy randomized adaptive search procedure for the feedback vertex set problem. Journal of Combinatorial Optimization, 2 (1999) 399–412
12. Pipatsrisawat, T.; Darwiche, A.: RSat Solver, version 1.03. http://reasoning.cs.ucla.edu/rsat/, accesessed in March 2007.
13. Smith, B.: Modelling. A chapter in Handbook of Constraint Programming, Elsevier (2006) 377–406
14. Stallman, R.M.; Sussman, G.J.: Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. Artificial Intelligence 9 (1997) 135–196

On the Integration of Singleton Consistency and Look-Ahead Heuristics

Marco Correia and Pedro Barahona

Universidade Nova de Lisboa, 2829-516 Caparica, Portugal
{mvc,pb}@di.fct.unl.pt

Abstract. The efficiency of complete solvers depends both on constraint propagation to narrow the domains and heuristics for directing search. Whereas constraint propagators should achieve a good trade-off between their complexity and the pruning that is obtained, heuristics take decisions based on information about the state of the problem being solved. In general, these two components are independent and are indeed considered separately. Nevertheless, a class of propagators have been proposed that maintain variants of Singleton Consistency (SC) performs look-ahead tests regarding the assignment of values from the domain of the variables, that can gather useful information, that is largely ignored. In this paper we discuss the integration of look-ahead information gathered while achieving SC into variable and value selection heuristics, and show that significant speed ups are obtained in a number of standard benchmark problems.

1 Introduction

Complete constraint programming solvers (i.e. relying on some form of complete backtracking, not incomplete local search) have their efficiency dependent on two complementary components, propagation and search. Constraint propagation is a key component in constraint solving, eliminating values from the domains of the variables with polynomial algorithms. When propagation is effective, such algorithms reduce the search space by some combinatorial (i.e exponential) factor, with a polynomial cost. The other component, search, aims at finding solutions in the remaining search space, and is usually driven by heuristics both for selecting the variable to enumerate and the value that is chosen first.

Typically, these components are independent. In particular, heuristics take into account some features of the remaining search space, and some structure of the problem to take decisions. Clearly, the more information there is, the more likely it is to get a good (informed) heuristics. In many search problems addressed in Artificial Intelligence, additional information is often obtained by performing a limited amount of look-ahead, and assessing the state of search some steps ahead. For example in two players games, like chess or chequers, rather than considering the current state of the board to decide the move to make, a number of moves by both players can be simulated, such that board

configurations with a more advanced phase of the game (thus better informed) can be taken into consideration.

Although better solver performances can be achieved by improving any of the two components (propagation and search) they are seldom, if ever, integrated. The main reason for this lies in that typical propagation procedures, such as maintenance of (generalised) arc and path consistency, are basically local filtering algorithms. Recently, a lot of attention has been given to a class of algorithms which analyse look-ahead what-if scenarios: what would happen if a variable x takes some value v? Such look-ahead analysis (typically done by subsequently maintaining arc or generalised arc consistency on the constraint network) may detect that value v is not part of any solution, and eliminate it from the domain of variable x. This is the purpose of the different variants of Singleton Consistency [7,1,4,14].

In this paper we propose to go one step further of the above approaches. On the one hand, by recognising that SAC propagation is not very cost-effective [16], we propose to restrict it to those variables more likely to be chosen by the variable selection heuristics. More specifically, we assume that there are often many variables that can be selected and for which no good criteria exists to discriminate them. This is the case with the first-fail (FF) heuristics, where often there are many variables with 2 values, all connected to the same number of other variables (as is the case with complete graphs). Hence the information gain obtained from SAC propagation is used to break the ties between the pre-selected variables.

On the other hand, we attempt to better exploit the information made available by the lookahead procedure, and use it not only to filter values but also to guide search.

The idea of exploiting look-ahead information is not new. However, and apart from AI search problems in which it has been used, in the context of Constraint Programming, look-ahead information has not been thoroughly exploited in subsequent variable or value selection heuristics. To the best of our knowledge it has only been exploited in [12], but as a complementary form of propagation: if the estimated likelihood of a value belonging to some solution decreases bellow some acceptable threshold, the value is discarded from the domain.

In this paper we thus investigate the possibility of integrating Singleton Consistency propagation procedures with look-ahead heuristics, both for variable and value selection heuristics, and analyse the speed ups obtained in a number of benchmark problems. The structure of the paper is the following. In the next section we review some algorithms for maintaining variants of Singleton Consistency, and show how to adapt them to obtain some look-ahead information. In section 3 we present a number of benchmark problems and compare the results obtained by using or not using the look-ahead heuristics. In the last section we conclude with a summary of the lessons learned and further research still to be done.

Algorithm 1 SC($\mathcal{X}$,$\mathcal{C}$) : *state*

```
do
   modified ←false
   forall x ∈ X
      modified ← sREVISE(x,X,C) ∨modified
      if D(x) = ∅
         state ←failed
         return
      endif
   endfor
while modified =true
state ←succeed
```

2 Look-ahead pruning algorithms

2.1 Formal background

A constraint network consists of a set of variables $\mathcal{X}$, a set of domains $\mathcal{D}$, and a set of constraints $\mathcal{C}$. Every variable $x \in \mathcal{X}$ has an associated domain $D(x)$ denoting its possible values. Every k-ary constraint $c \in \mathcal{C}$ is defined over a set of k variables $(x_1, \ldots, x_k)$ by the subset of the Cartesian product $D(x_1) \times \ldots \times D(x_k)$ which are consistent values. The constraint satisfaction problem (CSP) consists in finding an assignment of values to variables such that all constraints are satisfied.

A CSP is arc-consistent iff it has non-empty domains and every consistent instantiation of a variable can be extended to a consistent instantiation involving an additional variable [15]. A problem is generalized arc-consistent (GAC) iff for every value in each variable of a constraint there exist compatible values for all the other variables in the constraint.

A CSP P is singleton θ-consistent (SC), iff it has non-empty domains and for any value $a \in dom\,(x)$ of every variable $x \in \mathcal{X}$, the resulting subproblem $P|_{x=a}$ can be made θ-consistent. Cost-effective singleton consistencies are singleton arc-consistency (SAC) [7] and singleton generalized arc-consistency (SGAC) [16].

2.2 Look-ahead pruning algorithms

To achieve SC in a CSP, procedure SC [7] instantiates each variable to each of its possible values in order to prune those that (after some form of propagation) lead to a domain wipe out (alg. 1). Once some (inconsistent) value is removed, then there is a chance that a value in a previously revised variable has become inconsistent, and therefore SC must check these variables again. This can happen at most nd times, where n is the number of variables, and d the size of the largest domain, hence SC time complexity is in O($n^2d^2\Theta$), Θ being the time complexity of the algorithm that achieves θ-consistency on the constraint network. Variants of this algorithm with the same pruning power but yielding distinct space-time complexity trade-offs have been proposed [1,3,4,14]. A related algorithm considers each variable only once (alg. 2), has better runtime complexity O($nd\Theta$), but achieves a weaker consistency, called restricted singleton consistency (RSC) [16].

Algorithm 2 RSC($\mathcal{X}$,$\mathcal{C}$) : *state*

forall $x \in \mathcal{X}$
  SREVISE(x,$\mathcal{X}$,$\mathcal{C}$)
  if $D(x) = \emptyset$
    state ←**failed**
    return
  endif
endfor
state ←**succeed**

Algorithm 3 SREVISE(x,$\mathcal{X}$,$\mathcal{C}$) : $modified$

$modified$ ←**false**
forall $a \in D(x)$
  try $x = a$
    state ←PROPAGATE$_\theta$($\mathcal{X}$,$\mathcal{C}$)
  undo $x = a$
  if *state* =**failed**
    $D(x) \leftarrow D(x) \setminus a$
    $modified$ ←**true**
  endif
endfor

Note that both algorithms use function SREVISE (alg. 3) which prunes the domain of a single variable by trying all of its possible instantiations.

Another possible trade-off between run-time complexity and pruning power is to enforce singleton consistency on a subset of variables $\mathcal{S} \subset \mathcal{X}$. We identified two possible directions for the selection of $\mathcal{S}$. From a filtering perspective, $\mathcal{S}$ should be the smallest subset where (restricted) singleton consistency can actually prune values, but obviously this is not known *a priori*. On the other hand, variables in $\mathcal{S}$ may be selected for improving the underlying search procedure, in particular variable selection heuristics that are based on the cardinality of the current domains. In this case, the pruning resulting from enforcing singleton consistency is used as a mechanism to break ties both in the selection of variable and in the choice of the value to enumerate.

A large class of heuristics follow the first-fail/best-promise policy (FF/BP) [11], which consists in selecting the variable which more likely leads to a contradiction (FF), and then select the value that has more chances of being part of a solution (BP). For estimating first-failness, heuristics typically select the variable with smaller domain (dom), more constraints attached (deg), more constraints to instantiated variables (bdeg), or combinations (e.g. dom/deg). Best-promise is usually obtained by integrating some knowledge about the structure of the problem. Observing the general preference for variable heuristics which select smallest domains first, we propose defining $\mathcal{S}$ as the set of variables whose domain has cardinality below a given threshold.

Algorithm 4 sReviseInfo(x,$\mathcal{X}$,$\mathcal{C}$,$info$) : $modified$

```
modified ←false
forall a ∈ D(x)
   try x = a
      state ←PROPAGATE_θ(X,C)
      info[x, a] ← COLLECTINFO(X, C)
   undo x = a
   if state =failed
      D(x) ← D(x) \ a
      modified ←true
   endif
endfor
```

2.3 Extensions to look-ahead pruning algorithms

A further step in integrating singleton consistencies with search heuristics is to explore information regarding the subproblems that are generated each time a value is tested for consistency. We now propose a class of lookahead heuristics (LA) for any CSP P which reason over the size of its solution space, given by a function $\sigma(P)$, collected while enforcing singleton consistency. Although there is no known polynomial algorithm for computing σ (finding the number of solutions of a CSP is a #P-complete problem), there exists a number of naive and well as more sophisticated approximation functions [9,12]. We conjecture that by estimating the size of the solution space for each possible instantiation, i.e. $\sigma(P|_{x=a})$, there is an opportunity for making more informed decisions that will exhibit both better first-failness and best-promise behaviour. Moreover, the class of approximations of σ presented below are easy to compute, do not add complexity to the cost of generating the subproblems, and only requires a slight modification of the sRevise algorithm.

The sReviseInfo algorithm (alg. 4) stores in a table ($info$) relevant information to the specific subproblem being considered in each loop iteration. In our case, $info$ is an estimation of the subproblem solution space, more formally $info[x,a] = \sigma'(P|_{x=a})$ where $\sigma' \approx \sigma$. The table is initialized before singleton consistency enforcement, computed after propagation, and handed to the selectVariable and selectValue functions as shown in algorithm 5 . There are several possible definitions for these functions associated with how they integrate the collected information. Regarding the selection of variable for a given CSP P, we identified two FF heuristics which are cheap and easy to compute:

$$var_1(P) = \arg\min_{x \in \mathcal{X}(P)} \left(\sum_{a \in D(x)} \sigma'(P|_{x=a}) \right)$$

$$var_2(P) = \arg\min_{x \in \mathcal{X}(P)} \left(\max_{a \in D(x)} \sigma'(P|_{x=a}) \right)$$

Algorithm 5 SEARCH($\mathcal{X}$,$\mathcal{C}$) : *state*

$info \leftarrow \emptyset$
if SC($\mathcal{X}, \mathcal{C}$) =**fail**
  $state \leftarrow$**fail**
  return
endif
if $\forall_x : |D(x)| = 1$
  $state \leftarrow$**succeed**
  return
endif
$x \leftarrow$ SELECTVARIABLE($\mathcal{X}, info$)
$a \leftarrow$ SELECTVALUE($x, info$)
$state \leftarrow$ SEARCH($\mathcal{X}, \mathcal{C} \cup (x = a)$) **or** SEARCH($\mathcal{X}, \mathcal{C} \cup (x \neq a)$)

Informally, var_1 gives preference for the variable with a smaller sum of the number of solutions for every possible instantiation, while var_2 selects the variable whose instantiation with maximum number of solutions is the minimum among all variables. For the selection of value for some variable x, a possible BP heuristic is

$$val_1\left(P, x\right) = \arg \max_{a \in D(x)} \left(\sigma'\left(P|_{x=a}\right)\right)$$

which simply prefers the instantiation that prunes less solutions from the remaining search space.

Please note that we do not claim these are the best options for the estimation of the search space or the number of solutions. We have simply adopted them for simplicity and for testing the concept (more discussion on section 4).

3 Experimental results

A theoretical analysis on the validity of these heuristics as FF or BP candidates is needed, but hard to accomplish. Alternatively, in this section we attempt to give some empirical evidence of the quality of these heuristics by presenting the results of using them combined with constraint propagation and backtracking search (BT) on a set of typical CSP benchmarks.

The set of heuristics selected for comparison was chosen in order to provide some insight on the following questions:

- Is enforcing SC on a subset of variables a good trade-off between propagation and search?
- What is the potential gain on integrating LA information in the variable and value selection heuristics?

As a side effect, we tried to confirm previous results of on the following points:

- On which instances does SC payoff?

– When should RSC be preferred over SC, if ever?

We denote by $\text{RSC}_d(\mathcal{X}, \mathcal{C})$ and $\text{SC}_d(\mathcal{X}, \mathcal{C})$, respectively, the algorithms $\text{RSC}(\mathcal{X}_{|D|=d}, \mathcal{C})$ and $\text{SC}(\mathcal{X}_{|D|=d}, \mathcal{C})$, where $\mathcal{X}_{|D|=d}$ is the subset of variables in $\mathcal{X}$ having domains with cardinality d. On the remaining of this section we show the results of tests involving $d = 2$ only, since the most interesting results were obtained using this threshold.

As a first attempt at measuring the potential of LA heuristics, a simple measure was used for estimating the number of solutions in a given subproblem:

$$\sigma' = \sum_{x \in \mathcal{X}} \log_2 (D(x))$$

which informally expresses that the number of solutions is correlated to the size of the subproblem search space[1]. Although this is a very rough estimate, we are assuming that it could nevertheless provide valuable information to compare alternatives (see section 4).

As a baseline for comparison the dom variable selection heuristic was used without any kind of singleton consistency enforcing. The other elements of the test set are the possible combinations of enforcing SC, RSC, SC2 and RSC2 with the dom or LA heuristics.

In the following experiments all times are given in seconds, and represent the time needed for finding the first solution. The column 'ratio', when present, refers to the average CPU time of the current heuristic over the baseline, which is always the CPU time of the dom heuristic. Data presented in the following charts was interpolated using a bezier smoothing curve.

Tests regarding sections 3.1 and 3.2 were performed on a Pentium4, 3.4GHz with 1Gb RAM, while the results presented in section 3.3 were obtained on a Pentium4, 1.7GHz with 512Mb RAM.

3.1 Graph Coloring

Graph coloring consists of trying to assign n colors to m nodes of a given graph such that no pair of connected nodes have the same color. In this section we evaluate the performance of the presented heuristics in two sets of 100 instances of 10-colorable graphs, respectively with 50 and 55 nodes, generated using Joseph Culberson's k-colorable graph generator [6].

A CSP for solving the graph coloring problem was modelled by using variables to represent each node and values to define its color. Inequality binary constraints were posted for every pair of connected nodes.

The average degree of a node in the graph d, i.e. the probability that each node is connected to every other node, has been used for describing the phase transition in graph coloring problems [5]. In this experiment we started by determining empirically the phase transition to be near $d = 0.6$, and then generated 100 random instances varying d uniformly in the range $[0.5 \ldots 0.7]$.

[1] We use the logarithm since the size of search space can be a very large number.

Figure 1 compares the search effort using each heuristic on the smallest graph problem, with a timeout of 300 seconds. These results clearly divide the heuristics

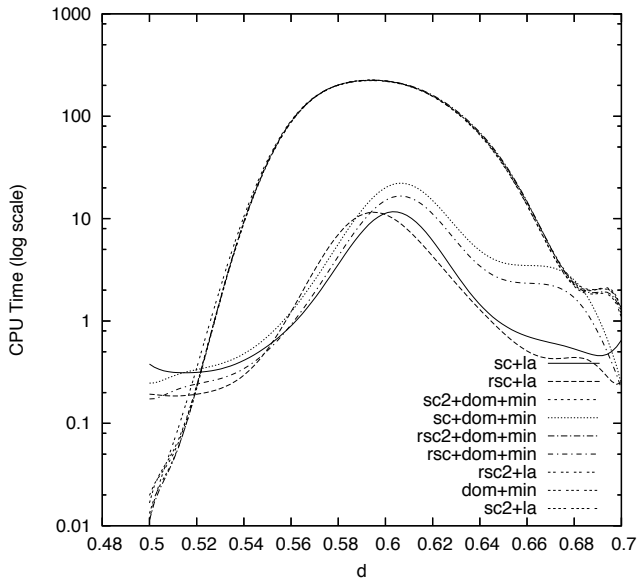


Fig. 1. CPU time spent in finding the first solution of random 10-colorable graph instances with size 50 .

in two sets, the set where SC and RSC was used being much better than the other on the hard instances. Since the ranking within the best set was not so clear, a second experiment on the larger and more difficult problem was performed using only these four heuristics, with a larger timeout of 900 seconds. The results of these tests are shown graphically on fig. 2, and also given in detail in table 1. This second set of experiments shows that RSC+LA is better on the most difficult instances (almost by an order of magnitude), while the others have quite similar efficiency.

3.2 Random CSPs

Randomly generated CSPs have been widely used experimentally, for instance to compare different solution algorithms. In this section we evaluate the lookahead heuristics on several random n-ary CSPs. These problems were generated using model C [10] generalized to n-ary CSPs, that is, each instance is defined by a 5-tuple $\langle n, d, a, p_1, p_2 \rangle$, where n is the number of variables, d is the uniform size of the domains, a is the uniform constraint arity, p_1 is the density of the constraint graph, and p_2 the looseness of the constraints.

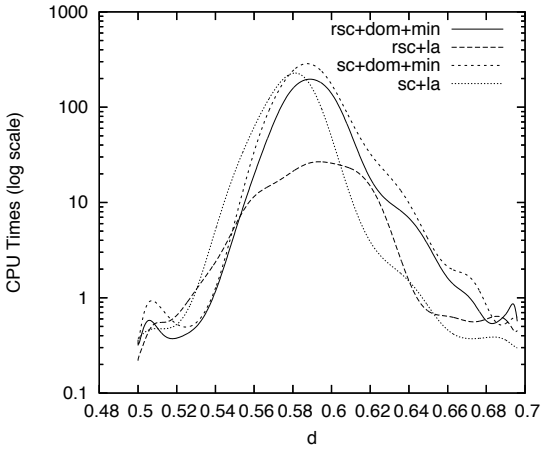


Fig. 2. CPU time spent in finding the first solution of random 10-colorable graph instances with size 55.

heuristic	d							
	0.500 0.525	0.525 0.550	0.550 0.575	0.575 0.600	0.600 0.625	0.625 0.650	0.650 0.675	0.675 0.700
rsc+dom+min	0.71	8.46	160.59	499.27	179.93	26.60	13.59	1.00
sc+dom+min	0.93	**0.83**	183.58	624.11	184.67	46.13	21.43	0.47
rsc+la	1.18	5.03	**148.92**	**67.13**	**72.62**	**26.44**	0.73	0.72
sc+la	**0.46**	104.29	320.35	603.55	107.27	69.51	**0.36**	**0.37**

Table 1. CPU time spent in finding the first solution of random 10-colorable graph instances with size 55. Columns show averages for intervals of uniform variation of constraint tightness d.

These tests evaluate the performance of the several heuristics in a set of random instances near the phase transition. For this task we used the constrainedness measure κ [9] for the case where all constraints have the same looseness and all domains have the same size:

$$\kappa = \frac{-|\mathcal{C}| \log_2 (p_2)}{n \log_2 d}$$

where $|\mathcal{C}|$ is the number of n-ary constraints.

We started by fixing n, d and a arbitrarily to 50, 5 and 3 respectively, and then computed 100 values for p_2 uniformly in the range $[0.1 \ldots 0.8]$. For each of these values, a value of p_1 was used such that $\kappa = 0.95$ (problems in the phase transition have typically $\kappa \approx 1$). The value of p_1, given by

$$p_1 = -\kappa \frac{n \log_2 d}{\log_2 p_2} \times \frac{a!\,(n-a)!}{n!}$$

is computed from the first formula and by noting that p_1 is the fraction of constraints over all possible constraints in the constraint graph, i.e.

$$p_1 = |\mathcal{C}| \frac{a!\,(n-a)!}{n!}$$

Solutions were stored as positive table constraints and GAC-Schema [2] was used for filtering. The timeout was set to 600 seconds.

Table 2 shows the results obtained. In figure 3 the performances of the most interesting heuristics are presented graphically. Besides the rsc+dom+min and

	p_2						
heuristic	0.1-0.2	0.2-0.3	0.3-0.4	0.4-0.5	0.5-0.6	0.6-0.7	0.7-0.8
dom+min	**0.06**	**0.34**	2.98	12.86	52.84	236.82	377.18
rsc2+dom+min	0.10	0.58	4.69	18.79	73.61	279.86	429.22
rsc+dom+min	0.21	1.09	5.35	25.03	97.32	319.71	471.83
sc2+dom+min	0.11	0.64	4.95	20.45	79.78	289.59	438.92
sc+dom+min	0.27	1.28	6.27	29.28	112.15	341.82	492.76
rsc2+la	**0.09**	0.45	3.47	11.85	53.64	237.19	373.80
rsc+la	0.11	**0.37**	**1.43**	**3.28**	**21.86**	**71.10**	**99.93**
sc2+la	0.10	0.50	3.78	12.92	58.60	249.03	388.77
sc+la	0.15	0.47	**1.75**	**4.04**	**26.06**	**82.60**	**115.13**

Table 2. CPU time spent in finding the first solution of random CSP instances Columns show averages for intervals of uniform variation of constraint looseness p_2.

sc+dom+min heuristics which always performed worse than the others, there seems to be a change of ranking around $p_2 \approx 0.4$, with the dom+min dominating on the dense instances, and LA heuristics 3-4 times faster on the sparse zone. RSC+LA and SC+LA are consistently close across all instances, being RSC slightly better.

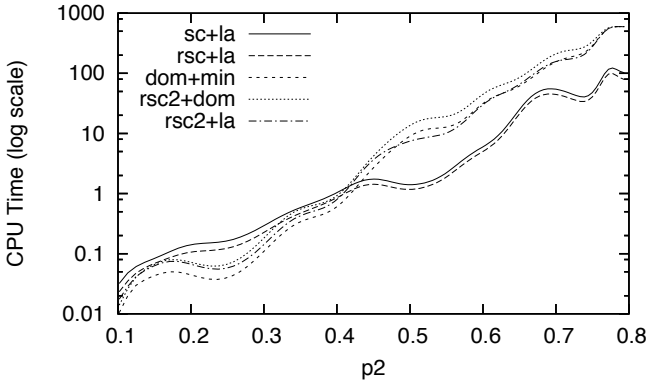


Fig. 3. CPU time spent in finding the first solution of random CSP instances.

3.3 Partial Latin Squares

Latin squares is a well known benchmark which combines randomness and structure [17]. The problem consists in placing the elements $1 \ldots N$ in a $N \times N$ grid, such that each element occurs exactly once on the same row or column. A partial Latin squares (or quasigroup completion) problem is a Latin squares problem with a number of preassigned cells, and the goal is to complete the puzzle.

The problem was modelled using the direct encoding, i.e. using an all-different constraint for every row and column. The dual encoding model, as proposed in [8], was also considered but never improved over the direct model on the presented instances. The value selection heuristic used in conjunction with the dom variable selection heuristic, denoted as mc (minimum-conflicts), selects the value which occurs less in the same row and column of the variable to instantiate. This is reported to be the best known value selection heuristic for this problem [8].

We generated 200 instances of a satisfiable partial Latin squares of size 30, with 312 cells preassigned, using lsencode-v1.1 [13], a widely used random quasigroup completion problem generator. The timeout was set to 900 seconds.

Results are presented on table 3. In this problem there is a clear evidence of the rsc2+la and sc2+la heuristics compared to every other. Besides the fact that they are over 5 times faster than the other alternatives, they are also the most robust, as shown by their lower standard deviations as well as the absence of time out instances.

3.4 Discussion

The results obtained clarified some of the questions posed in the beginning of this section. In particular, the best performing combinations in all problems were always obtained using LA information, so this approach has clearly some potential to be explored more thoroughly.

		#fails		time		
heuristic	#timeouts	avg	std	avg	std	ratio
dom+mc	5	18658	43583	66.7	169	1
rsc2+dom+mc	5	235	436	70.2	156.4	1.05
rsc+dom+mc	5	24	49	89.3	159.2	1.34
sc2+dom+mc	10	174	330	100.5	207.1	1.51
sc+dom+mc	6	**15**	**28**	122.8	180.3	1.84
rsc2+la	**0**	51	127	**12.2**	**19.9**	**0.18**
rsc+la	**0**	14	35	67.7	47.7	1.02
sc2+la	**0**	43	109	**15.6**	**26.4**	**0.23**
sc+la	4	**11**	**29**	104.6	134.4	1.57

Table 3. Running times and number of fails for the pls-30-312 problem. Last column shows the ratio between the average time of each heuristic over the average time of the baseline, which is the dom+mc heuristic.

Regarding the question of using SC on a subset of variables, the results so far are not conclusive. Heuristics that restrict SC maintenance to only 2 valued variables performed badly both on the graph coloring and random problems, and clearly outperformed all others on the Latin squares problem. We think that this behaviour may be connected with the number of times these heuristics have a chance to break ties both in the selection of variable and value. The cardinality of the domains should have impact on the number of decisions having the same preference for FF heuristics, in particular the dom heuristic. The average number of values in the Latin squares problem is very low (around 3) since most variables are instantiated, so these heuristics would have more chance to make a difference here than on the other problems which have larger cardinality (5 and 10). The same argument may apply to the value selection heuristic if we note that the selection of value is more important in problems with structure, which would again favour the Latin squares problem.

The remaining aspects of the results obtained are in accordance with the extensive analysis of singleton consistencies described in [16]. On the question of cost-effectiveness of RSC we obtained similar positive results, in fact it was slightly better than SC on all instances. Its combination with LA was the most successful, outperforming the others in the hard instances of every problem.

In the class of random problems, their work concludes that singleton consistencies are only useful in the sparse instances. Our results also confirms this. Generally, the claim that SC can be very expensive to maintain seems true in our experiments except when using combined with LA heuristics. This provides some evidence that the good behaviour of SC+LA observed relies more strongly on correct decisions rather than on the filtering achieved.

4 Conclusion

In this paper we presented an approach that incorporates look ahead information for directing backtracking search, and suggested that this could be largely done at no extra cost by taking advantage of the work already performed by singleton consistency enforcing algorithms. We described how such a framework could extend existing SC and RSC algorithms by requiring only minimal modifications. Additionally, a less expensive form of SC was revisited, and a new one proposed which involves revising only a subset of variables. Empirical tests with 2 typical benchmarks and with randomly generated CSPs showed promising results on instances near the phase transition. Finally, results were analysed and matched against those previously obtained by other researchers.

There are a number of open questions and future work directions. As discussed in the previous section, tests which use singleton consistency on a subset of variables defined by its cardinality were not consistently better or worse than the others, but may be very beneficial sometimes. We think this deserves more investigation, namely testing with more structured problems, and using a distinct selection criteria (other than domain cardinality).

The most promising direction for future work is perhaps to improve the FF and BP measures. Lookahead heuristics presented above use rather naive estimation of number of solutions for a given subproblem compared to, for example, the κ measure introduced in [9], or the probabilistic inference methods described in [12]. The κ measure, for example, takes into account the individual tightness of each constraint and the global density of the constraint graph. Their work shows strong evidence for best performance of this measure compared with standard FF heuristics, but also point out that the complexity of its computation may lead to suboptimal results in general CSP solving (the results reported are when using forward-checking). Given that we perform a stronger form of propagation and have lookahead information available, the cost for computing κ may be worth while. We intend to investigate this in the future.

Other improvements include a) the use of faster singleton consistency enforcing algorithms [1,4], which should be orthogonal to the results presented here, and b) the use of constructive disjunction during the maintenance of SC, by pruning values from the domains of a variable that does not appear in the state of the problem for all values of another variable.

Notwithstanding the limitations of our current implementation, the results obtained so far are quite promising and justify further research along the outlined directions.

References

1. Roman Barták and Radek Erben. A new algorithm for singleton arc consistency. In Valerie Barr and Zdravko Markov, editors, *Proceedings of the Seventeenth International Florida Artificial Intelligence Research Society Conference (FLAIRS'04), Miami Beach, Florida, USA*. AAAI Press, 2004.

2. C. Bessiére and J-C. Régin. Arc consistency for general constraint networks: preliminary results. In *Proceedings of IJCAI'97*, pages 398–404, Nagoya, Japan, 1997.
3. Christian Bessière and Romuald Debruyne. Theoretical analysis of singleton arc consistency. In *Proceedings of ECAI'04*, 2004.
4. Christian Bessière and Romuald Debruyne. Optimal and suboptimal singleton arc consistency algorithms. In *Proceedings of IJCAI'05*, 2005.
5. Peter Cheeseman, Bob Kanefsky, and William M. Taylor. Where the really hard problems are. In *Proceeding of IJCAI'91*, pages 331–340, 1991.
6. Joseph Culberson. Graph coloring resources. on-line. http://web.cs.ualberta.ca/ joe/Coloring/Generators/generate.html.
7. Romuald Debruyne and Christian Bessière. Some practicable filtering techniques for the constraint satisfaction problem. In *Proceedings of IJCAI'97*, pages 412–417, 1997.
8. Dotu, del Val, and Cebrian. Redundant modeling for the quasigroup completion problem. In *ICCP: International Conference on Constraint Programming (CP), LNCS*, 2003.
9. I. P. Gent, E. MacIntyre, P. Prosser, and T. Walsh. The constrainedness of search. In *Proceedings of AAAI'96*, volume 1, pages 246–252, 1996.
10. Ian P. Gent, Ewan MacIntyre, Patrick Prosser, Barbara M. Smith, and Toby Walsh. Random constraint satisfaction: Flaws and structure. *Constraints*, 6(4):345–372, 2001.
11. R. M. Haralick and G. L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980.
12. Kalev Kask, Rina Dechter, and Vibhav Gogate. New look-ahead schemes for constraint satisfaction. In *Proceeding of AMAI'04*, 2004.
13. Kautz, Ruan, Achlioptas, Gomes, Selman, and Stickel. Balance and filtering in structured satisfiable problems. In *Proceedings of IJCAI'01*, 2001.
14. Christophe Lecoutre and Stéphane Cardon. A greedy approach to establish singleton arc consistency. In Leslie Pack Kaelbling and Alessandro Saffiotti, editors, *Proceedings of IJCAI-05*, pages 199–204. Professional Book Center, 2005.
15. Alan K. Mackworth and Eugene C. Freuder. The complexity of some polynomial network consistency algorithms for constraint satisfaction problems. *Artificial Intelligence*, 25:65–74, 1985.
16. Patrick Prosser, Kostas Stergiou, and Toby Walsh. Singleton consistencies. In Rina Dechter, editor, *Proceeding of CP'00*, volume 1894 of *Lecture Notes in Computer Science*, pages 353–368. Springer, 2000.
17. Paul Shaw, Kostas Stergiou, and Toby Walsh. Arc consistency and quasigroup completion. In *Proceedings of ECAI'98 workshop on non-binary constraints*, March 14 1998.

Projection Global Consistency: An Application in AI Planning

Pavel Surynek

Charles University
Faculty of Mathematics and Physics
Malostranské náměstí 2/25, 118 00 Praha 1, Czech Republic
pavel.surynek@mff.cuni.cz

Abstract. We are dealing with solving planning problems by the GraphPlan algorithm. We concentrate on solving a problem of finding supporting actions for a goal. This problem arises as a sub-problem many times during search for a solution. We showed in this paper that the supports problem is NP-complete. In order to improve the solving process of supports problems we proposed a new global consistency technique which we call a projection consistency. We present a polynomial algorithm for enforcing the projection consistency. The projection consistency was implemented within our experimental planning system that was used for empirical evaluation. The performed empirical tests showed improvements in order of magnitudes compared to the standard GraphPlan. A significant improvement was also reached compared to the recent technique based on maintaining of arc-consistency which solves the same problem.

1 Introduction

In this paper, we would like to explain our new contribution to solving AI planning problems. We called our new concept a *projection global consistency*. It is designed to prune the search space during solving planning problems over planning graphs by the GraphPlan-style planning algorithm.

Planning as a task of finding a sequence of actions resulting in achieving some goal is one of the most challenging problems of artificial intelligence [2]. The need of solving planning problems arises almost every time when a complex autonomous behavior of a certain agent is required. It is the case of spacecrafts and vehicles for distant space and planetary exploration [1, 3] as well as the case of unmanned military devices [5]. There are many successful algorithmic techniques for solving planning problems. One of them is usage of so called *planning graphs* on which we concentrate in this paper. The concept of planning graphs introduced by Blum and Furst [4] brought a substantial break-through in solving of planning problems. Many of the consequent achievements in planning are based on ideas of planning graphs. In this paper we are studying planning graphs from the perspective of constraint programming [6]. We analyzed the original Blum's and Furst's GraphPlan algorithm [4] as well as other approaches based on that [9, 10, 11, 13]. Our conclusion was that there

is a room for exploiting some type of a global reasoning which is in constraint programming known as *global constraints* [15].

The paper is organized as follows. First we introduce basic definition of planning problems and also some definitions from constraint programming. Then we briefly explain planning graphs. The next section is devoted to a sub-problem which arises during the search using GraphPlan algorithm. The consequent main part of the paper describes the projection consistency and projection constraint which are designed to help to solve the mentioned sub-problem. Finally we present some empirical tests of the proposed concept and discuss our contribution in relation to other works.

2 AI Planning and Constraint Programming Essentials

For purposes of clarity we are using a simple language for expressing planning problems in this paper. A language is associated with a planning domain. To describe a problem over a certain planning domain we use language L with finitely many predicate and constant symbols. The set of predicates will be denoted as P_L and the set of constants as C_L. Constants represent objects appearing in the planning world and predicate symbols are used to express relations over objects. Let us note that simplicity of the language is not at the expense of expressivity (in [8] Ghallab et al. show more approaches for describing planning problems). The following definitions assume a fixed language L.

Definition 1 (Atom). An *atomic formula* (atom) is a construct of the form $p(c_1,c_2,\ldots,c_n)$ where $p \in P_L$ and $c_i \in C_L$ for $i = 1,2,\ldots,n$.

Definition 2 (State, Goal, Goal satisfaction). A *state* is a finite set of atoms. A *goal* is also a finite set of atoms. The goal g is *satisfied* in the state s if $g \subseteq s$.

States provide a formal description of a situation in the planning world. A goal is a formal description of a situation which we want to establish. The situation in the planning world is changed by actions. Actions formally define possible transitions between states. Action applied to the state results into a new state.

Definition 3 (Action, Applicability, Application). An *action* a is a triple $(p(a),e^+(a),e^-(a))$, where $p(a)$ is a *precondition* of the action, $e^+(a)$ is a *positive effect* of the action and $e^-(a)$ is a *negative effect* of the action. All the three components of the action are finite sets of atoms. An action a is *applicable* to the state s if $p(a) \subseteq s$. The *result of the application* of the action a to the state s is a new state $\gamma(s,a)$, where $\gamma(s,a) = (s - e^-) \cup e^+$.

For purposes of planning graphs there are also assumed so called *no-op actions*. For every atom t we assume a no-op action $noop_t = (t,t,\emptyset)$. Briefly said a no-op action preserves an atom into the next state.

Given a set of allowed actions and a goal the objective is to transform a given initial state into a state satisfying the goal. State transitions to achieve the objective are carried out by applying actions from the set of allowed actions. We suppose that the number of preconditions, the number of positive effects and the number of negative effects are bounded by a constant.

Definition 4 (Planning problem). A *planning problem* P is a triple (s_0, g, A), where s_0 is an *initial state*, g is a goal and A is a finite set of allowed actions.

Definition 5 (Application of sequence of actions, Solution). We inductively define *application of a sequence of actions* $\phi = [a_1, a_2, \ldots, a_n]$ to a state s_0 in the following way: a_1 must be applicable to s_0, let us inductively denote the result of application of the action a_i to the state s_{i-1} as s_i for all $i = 1,2,\ldots,n$; the condition that a_i is applicable to the state s_{i-1} for all $i = 2,3,\ldots,n$ must hold. The *result of application of the sequence of actions* ϕ to the state s_0 is the state s_n. Sequence $\xi = [a_1, a_2, \ldots, a_n]$ is a *solution* of the planning problem $P = (s_0, g, A)$ if the sequence ξ is applicable to the initial state s_0 and the goal g is satisfied in the result of application of the sequence ξ and $a_i \in A$ for all $i = 1,2,\ldots,n$.

A method developed in this paper regards some problems from the constraint programming perspective. Therefore some basic definitions from constraint programming are necessary. The key concept is a *constraint satisfaction problem*.

Definition 6 (Constraint satisfaction problem) [6]. A *constraint satisfaction problem* (CSP) is a triple (X, D, C), where X is a finite set of variables, D is a finite domain of values for variables from the set X and C is a finite set of constraints. A *constraint* is an arbitrary relation over the domains of its variables. A sequence of variables constrained by a constraint $c \in C$ is denoted as X_c.

Definition 7 (Solution of CSP) [6]. A *solution* of a constraint satisfaction problem (X, D, C) is an assignment of values to the variables $\psi : X \to D$ such that all the constraints are satisfied for ψ, that is $(\forall c \in C)[x_1, x_2, \ldots, x_k] = X_c \Rightarrow [\psi(x_1), \psi(x_2), \ldots \ldots, \psi(x_k)] \in C$.

CSPs are often solved using so called *constraint propagation*. If a domain of a variable is changed then this change is propagated into the domains of other variables through constraints. The quality of the solving algorithm is often tightly connected with the quality of propagation techniques for individual constraints. It is especially true for so called *global constraints* which bind large number of variables and perform fine grained and effective propagation throughout the large parts of the problem (for example Régin's *allDifferent* constraint [15]). Our method is trying to follow the concept of such global propagation.

3 Planning Graphs and GraphPlan Algorithm

The *GraphPlan* algorithm is due to Blum and Furst [4]. It relies on the idea of state reachability analysis. The state reachability analysis is done by constructing a special data structure called *planning graph*. The algorithm itself works in two interleaved phases. In the first phase planning graph is incrementally expanded. Then in the second phase an attempt to extract a valid plan from the extended planning graph is performed. The GraphPlan algorithm uses standard backtracking to extract plan from planning graphs. If the second phase is unsuccessful the process continues with the first phase. That is the planning graph is extended again.

The planning graph for a planning problem $P = (s_0, g, A)$ is defined as follows. It consists of two alternating structures called *proposition layer* and *action layer*. The

initial state s_0 represents the *0*th proposition layer P_0. The layer P_0 is just a list of atoms occurring in s_0. The rest of the planning graph is defined inductively. Consider that the planning graph with layers P_0, A_1, P_1, A_2, P_2, …, A_k, P_k has been already constructed (A_i denotes the *i*th action layer, P_i denotes the *i*th proposition layer). The next action layer A_{k+1} consists of actions whose preconditions are included in the *k*th proposition layer P_k and which satisfy the additional condition that no two preconditions of the action are *mutually excluded* (we briefly say that they are *mutex*). The next proposition layer P_{k+1} consists of all the positive effects of actions from A_{k+1} (this is the reason for having no-op actions).

Definition 8 (Independence of actions). A pair of actions $\{a,b\}$ is *independent* if $e^-(a) \cap (p(b) \cup e^+(b)) = \varnothing$ and $e^-(b) \cap (p(a) \cup e^+(a)) = \varnothing$. Otherwise $\{a,b\}$ is a pair of *dependent* actions.

Definition 9 (Action mutex and mutex propagation). We call a pair of actions $\{a,b\}$ within the action layer A_i a *mutex* if either the pair $\{a,b\}$ is dependent or an atom of the precondition of the action a is mutex with an atom of the precondition of the action b (defined in the following definition).

Definition 10 (Proposition mutex and mutex propagation). We call a pair of atoms $\{p,q\}$ within the proposition layer P_i a *mutex* if every action a within the layer A_i where $p \in e^+(a)$ is mutex with every action b within the action layer A_i for which $q \in e^+(b)$ and the action layer A_i does not contain any action c for which $\{p,q\} \subseteq e^+(c)$.

4 Problem of Finding Supports for a Sub-goal

A problem of finding supports for a sub-goal is definable for arbitrary action layer of the planning graph and for arbitrary goal. Consider an action layer of a given planning graph. Let A be a set of actions of the action layer and let μA be a set of mutexes between actions from A. Next let us have a goal g. For the given goal g and the action layer A the question is to determine a set of actions $\zeta \subseteq A$ where no two actions from ζ are mutex with respect to μA and ζ satisfies the goal g. The set of actions ζ satisfies the goal g if $g \subseteq \bigcup_{a \in \zeta} e^+(a)$. The actions from the set ζ are called *supports* for the goal g in this context. The goal g is called a *sub-goal* to distinguish it from the global goal for which we are building a plan. Typically many sub-goals must be satisfied along the search for the global goal in the standard GraphPlan algorithm. The problem of finding supports for a sub-goal will be called a *supports problem* in short.

The effectiveness of a method for solving supports problem has a major impact on the performance of the planning algorithm as a whole. Unfortunately the supports problem is NP-complete. This claim can be easily proved by using reduction of Boolean formula satisfaction problem (SAT) to supports problem.

Theorem 1 (Complexity of supports problem). *The problem of finding supports for a sub-goal within an action layer of the planning graph is NP-complete.*

Proof: Observe that the supports problem is in *NP*. It is sufficient treat sets as lists to prove the claim. Checking if no two actions from ζ are mutex takes polynomial time. The question whether $g \subseteq \bigcup_{a \in \zeta} e^+(a)$ can be answered in polynomial time too.

Completeness with respect to *NP* class can be proved by using polynomial reduction of Boolean formula satisfaction problem (*SAT*) to supports problem. Consider a Boolean formula B. It is possible to assume that the formula B is in the form of conjunction of disjunctions, that is $B = \wedge_{i=1}^{n} \vee_{j=1}^{m_i} x_j^i$, where x_j^i is a variable or a negation of a variable. For each clause $\vee_{j=1}^{m_i} x_j^i$ where $i = 1,2,\ldots,n$ we introduce an atom t_i into the constructed goal g. Next we introduce an action $a_j^i = (\varnothing, \{t_i\}, \varnothing)$ into the set of actions A for each x_j^i from the clause. Actions are introduced in this way for all the clauses from B. If for some $i,k \in \{1,2,\ldots,n\}$; $j \in \{1,2,\ldots,m_i\}$; $l \in \{1,2,\ldots,m_k\}$ $x_j^i = \neg x_l^k$ or $\neg x_j^i = x_l^k$ holds we introduce a mutex $\{a_j^i, a_l^k\}$ into the set of mutexes μA. The constructed sets A, μA (action layer) and the goal g constitute the instance of the supports problem. The size of the resulting supports problem is $O(|B|^2)$, where $|B|$ is the number of literals appearing in B.

Having a set of actions ζ which solve the constructed instance of the supports problem we can construct valuation f as follows $f(x_j^i) = true$ for each $a_j^i \in A$ (that is, if $x_j^i = v$ for some variable v then $f(v) = true$, if $x_j^i = \neg v$ then $f(v) = false$). The truth values for the remaining variables in B can be selected arbitrarily. Mutexes ensure that the valuation f is correctly defined function. Moreover we have $f(\vee_{j=1}^{m_i} x_j^i) = true$ for $i = 1,2,\ldots,n$. Thus every clause of B is positively valued. This is implied by the fact that the whole goal g is satisfied by ζ. The solution of the original Boolean formula satisfaction problem is obtained from ζ in $O(|B|)$ steps. ■

5 Projection Constraint and Projection Consistency

We proposed a new global constraint to improve the search of the GraphPlan planning algorithm in the phase of plan extraction from the planning graph. We called the constraint a *projection constraint*. This name should convey the fact that this constraint is based on propagation for sub-goals (subsets of the goal - projections). We use the projection constraint to model and to improve the solving of the supports problem. The supports problem must be solved many times along the search for a plan in plan extraction phase. A part of the GraphPlan algorithm which solves the supports problem is responsible for majority of backtracks. That is why the efficiency of solving of this sub-problem has a significant impact on the overall efficiency.

In [17] and [18] Surynek examined the effect of maintaining arc-consistency and singleton arc-consistency for solving the supports problem. He obtained substantive speedups using these types of reasoning compared to pure backtracking based method. However both arc-consistency and singleton arc-consistency provides only some type of a local reasoning over the problem. Such feature of consistency technique may lead to high number of iterations of the consistency enforcing algorithm till the consistency is established. Moreover if the cost of an iteration of such local technique is too high there remains only a little advantage against the pure backtracking (namely this is the case of maintaining singleton arc-consistency [18]).

By contrast the projection constraint introduces some type of a global reasoning over the supports problem. It was motivated by observation of mutex graphs of layers of the planning graph. These mutex graphs embody high density of edges on majority of testing planning problems (however our method works with sparse mutex graphs as well). The high density of edges is caused by various factors. Nevertheless we regard the set of actions that change states of an object or group of objects of the planning domain as the most important one. Actions from such set are pair wise mutually excluded since they change a single property (for example imagine a robot at coordinates $[3,2]$, the robot can move to coordinates in its neighborhood, so the actions are: $moveTo([2,2])$, $moveTo([2,3])$, $moveTo([3,3])$, …). That is such set of actions induces a clique within a mutex graph.

The knowledge of clique decomposition of the mutex graph would allow us to identify the above described strong correspondence among actions from a clique (at most one action from a clique can be selected). Such knowledge can be used for some kind of advanced reasoning afterwards. This is just the first part of the idea how projection constraint works. The second part of the idea of projection constraint is to take a subset of atoms of a given goal and to calculate how a certain clique of actions contributes to satisfaction of the subset of atoms. This reasoning can be used to discover that some actions within a certain clique do not contribute enough to the goal and therefore can be ruled out. Actions that are ruled out are no more considered along the search and hence the search speeds up since smaller number of alternatives must be considered.

Projection constraint assumes that a clique decomposition of a mutex graph of a given action layer of the planning graph is known. Thus we need to perform a preprocessing step in which a clique decomposition (clique cover) of the mutex graph is constructed. Let $G = (A, \mu A)$ be a mutex graph (vertexes are represented by actions, edges are represented by mutexes). The task is to find a partition of the set of vertexes $A = C_1 \cup C_2 \cup \ldots \cup C_n$ such that $C_i \cap C_j = \emptyset$ for every $i, j \in \{1,2,\ldots,n\}$ & $i \neq j$ and C_i is a clique with respect to μA for $i = \{1,2,\ldots,n\}$. Cliques of the partitioning do not cover all the mutexes in general case. For $mA = \mu A - (C_1^2 \cup C_2^2 \cup \ldots \cup C_n^2)$ $mA \neq \emptyset$ holds in general (where $C^2 = \{\{a,b\} \mid a,b \in C \;\&\; a \neq b\}$). Our requirement is to minimize n and $|mA|$. Unfortunately the problem of clique cover of the defined property is obviously *NP*-complete on a graph without any restriction.

As an exponential amount of time spent in preprocessing step is unacceptable it is necessary to abandon the requirement on optimality of clique cover. It is sufficient to find some clique cover to be able to introduce projection constraint. However the better the clique cover is (with respect to n and $|mA|$) the better is the performance of projection constraint. Our experiments showed that a simple greedy algorithm provides satisfactory results. Its complexity is polynomial in size of the input graph which is acceptable for preprocessing step.

For the following description of projection constraint consider an action layer of the planning graph for which a clique cover $A = C_1 \cup C_2 \cup \ldots \cup C_n$ of the set of actions A with respect to the set of mutexes μA was computed. Projection consistency is defined over the above decomposition for a goal p. The goal p is called a *projection goal* in this context. The fact that at most one action from a clique can be selected allows us to introduce the following definition.

Definition 11 (Clique contribution). A *contribution* of a clique $C \in \{C_1, C_2, \ldots, C_n\}$ to the projection goal p is defined as $\max(|e^+(a) \cap p| \mid a \in C)$. The contribution of the clique C to the projection goal p is denoted as $c(C, p)$.

The concept of clique contribution is helpful when we are trying to decide whether it is possible to satisfy the projection goal using the actions from the clique cover. If for instance $\sum_{i=1}^{n} c(C_i, p) < |p|$ holds then the projection goal p cannot be satisfied. Nevertheless the projection constraint can handle a more general case as it is described in the following definitions.

Definition 12 (Projection consistency: supported action). An action $a \in C_i$ for $i \in \{1,2,\ldots,n\}$ is *supported* with respect to *projection consistency* with the projection goal p if $\sum_{j=1, j\neq i}^{n} c(C_j, p) \geq |p - e^+(a)|$ holds.

Definition 13 (Projection consistency: consistent problem). The preprocessed instance of the supports problem consisting of actions $A = C_1 \cup C_2 \cup \ldots \cup C_n$, mutexes μA and the goal g is *projection consistent* with respect to a projection goal $p \subseteq g$, $p \neq \emptyset$ if every action $a \in C_i$ for $i = 1,2,\ldots,n$ is supported.

If cliques of the clique cover are regarded as CSP variables and actions from the cliques are regarded as values for these variables then we can introduce a *projection constraint*. The projection constraint bounds domains of all the clique variables. That is the constraint bounds all the variables of the CSP problem. The constraint with respect to the projection goal $p \subseteq g$ is satisfied for an assignment $[C_1 = a_1, C_2 = a_2, \ldots, C_n = a_n]$ if $p \subseteq \bigcup_{i=1}^{n} e^+(a_i)$. To enforce projection consistency over the supports problem for some projection goal p we can easily remove values from the domains of variables. Specifically it is necessary to rule out actions which are not supported according to the definition 12 for the projection goal p. But notice that projection consistency is not a sufficient condition to obtain a solution. There still remain assignments for which the constraint is not satisfied. The second note is on the slight difference of the definition of a solution of the constraint satisfaction problem over the clique variables from the standard definition. We do not necessarily need to assign all the clique variables to solve the problem. The solution requires satisfaction of the projection goal only.

Proposition 1 (Correctness of projection consistency). Projection consistency is correct. That is the set of solutions of the supports problem S is the same as the set of solutions of the supports problem S' which we obtain from S by enforcing projection consistency with respect to a projection goal $p \subseteq g$.

Proof: The proposition is easy to prove by observing that an unsupported action cannot participate in any assignment satisfying the projection constraint for the projection goal p. Let $a \in C_i$ be an unsupported action for $i \in \{1,2,\ldots,n\}$, that is $\sum_{j=1, j\neq i}^{n} c(C_j, p) < |p - e^+(a)|$ holds. It is obvious that after the selection of the action a there is no chance to satisfy the goal p. So much the less chance to satisfy g. ■

A useful property of the projection consistency with a single projection goal p is that the removal of an unsupported action does not affect any of the remaining supported actions. That is if an action is supported, it remains supported after removal of any other unsupported action. We call this property a *monotonicity*. The usefulness consist in the fact that it is enough to check each action of the problem only once to enforce the projection consistency.

Proposition 2 (Monotonicity of projection consistency). Projection consistency with a projection goal p is *monotone*. That is if an arbitrary unsupported a action is removed from a clique C_i for $i \in \{1,2,\ldots,n\}$ the set of supported actions within the problem remains unchanged.

Proof: Let $b \in C_j$ be an unsupported action after removal of a from C_i. That is after removal of a from C_i $\sum_{k=1,k\neq j}^{n} c(C_k,p) < \left|p - e^+(b)\right|$ holds. First let us investigate the case when $i = j$. It is obvious that removal of a has no effect on the truth value of the expression $\sum_{k=1,k\neq j}^{n} c(C_k,p) < \left|p - e^+(b)\right|$. Hence the action b was unsupported even before a was removed. For the case when $i \neq j$ the situation is similar. If $c(C_i,p) = c(C_i - \{a\},p)$ then the removal of the action a has no effect on the truth value of the expression $\sum_{k=1,k\neq j}^{n} c(C_k,p) < \left|p - e^+(b)\right|$. If $c(C_i,p) > c(C_i - \{a\},p)$, then $\left|p \cap e^+(a)\right| = c(C_i,p)$. From the assumption $\sum_{j=1,j\neq i}^{n} c(C_j,p) < \left|p - e^+(a)\right|$ we have $\sum_{k=1}^{n} c(C_k,p) < \left|p\right|$. Hence also $\sum_{k=1,k\neq j}^{n} c(C_k,p) < \left|p - e^+(b)\right|$ holds. ■

In order to be able to discuss complexity issues of our approach we have to formally define propagation algorithm for projection consistency. The propagation algorithm for projection consistency is shown as algorithm 1. The input of the algorithm is a projection goal p and the clique decomposition.

Algorithm 1: Projection consistency propagation algorithm

function *propagateProjectionConsistency* $(\{C_1,C_2,\ldots,C_n\},p)$: **set**
1: $\gamma \leftarrow 0$
2: **for** $i = 1,2,\ldots,n$ **do**
3: | $c_i \leftarrow$ *calculateCliqueContribution* (C_i,p)
4: | $\gamma \leftarrow \gamma + c_i$
5: **for** $i = 1,2,\ldots,n$ **do**
6: | **for each** $a \in C_i$ **do**
7: | | **if** $\gamma + \left|e^+(a) \cap p\right| < \left|p - e^+(a)\right| + c_i$ **then** $C_i \leftarrow C_i - \{a\}$
8: **return** $\{C_1,C_2,\ldots,C_n\}$

function *calculateCliqueContribution* (C,p) : **integer**
9: $c \leftarrow 0$
10: **for each** $a \in C$ **do**
11: | $c \leftarrow \max(c, \left|e^+(a) \cap p\right|)$
12: **return** c

Theorem 2 (Complexity of projection consistency). *Propagation algorithm for projection consistency with a projection goal* $p \subseteq g$ *over the supports problem consisting of actions* $A = C_1 \cup C_2 \cup \ldots \cup C_n$, *mutexes* μA *and a goal* g *runs in* $O(|p||A|)$ *steps.*

Proof: Since the algorithm for enforcing projection consistency is quite straightforward it is easy compute its complexity. The auxiliary function *calculateCliqueContribution* performs $O(|p||C|)$ steps (the loop on lines 9-11 performs exactly $|C|$ iterations, each iteration of the loop takes $d|p|$ steps, where d is the action size bounding constant). Hence lines 2-4 of the main function *propagateProjectionConsistency* takes $O(\sum_{i=1}^{n} |p||C|) = O(|p||A|)$. Finally lines 5-7 of the main function performs a conditional statement on the line 7 $|A|$ times. Each check of the condition in the conditional

statement on the line 7 takes $d|p|$ steps. Hence we have a total number of steps in $O(|p||A|)$. ■

We were not concerned about the question of how to select projection goals for a problem with a goal g until now. The only condition on a projection goal p is that $p \subseteq g$ must hold.

The projection consistency filters out different sets of inconsistent actions for different projection goals. So it is suitable to enforce projection consistency with respect to several projection goals. For maximal pruning power, we would have to check projection consistency for every subset the goal g. But this is unaffordable since those are too many. Hence we can select only a limited number of projection goals. At the same time the selection must be done carefully in order to achieve strongest possible filtering effect. We provide a brief analysis of projection goal selection here. The following ideas are focused on comparison of projection consistency with arc-consistency of the supports problem as it was introduced in [17, 18].

Definition 14 (Arc-consistency of the supports problem) [17, 18]. Let us have a supports problem S with a goal g. For each atom $t \in g$ we introduce a so called support variable which contains all the actions that supports the atom t in its domain (an action a *supports* an atom t if $t \in e^+(a) \,\&\, t \notin e^-(a)$, a set $s_t = \{a \mid a \in A \,\&$ supports atom $t\}$ is called a s*et of support*s for an atom t). Between every two support variables there is a mutex constraint. The mutex constraint is satisfied by an assignment of actions to its variables if the actions of the assignment are non-mutex. The supports problem is *arc-consistent* if every mutex constraint is arc-consistent [14].

Depending on the quality of the clique decomposition of the mutex graph of the supports problem there may be a situation in which a projection goal can be selected to simulate arc-consistency by projection consistency. Moreover there may be situations when projection consistency is stronger than arc-consistency. Both cases are formally summarized in the following observations. Experiments showed that such cases are not rare, especially when projection goals are selected in order to prefer such cases.

Observation 1 (Arc-consistency by projection consistency). For a given supports problem S with a goal g there may be a projection goal $p \subseteq g$ such that if the problem S is projection consistent with the projection goal p then it is arc-consistent.

Proof: It is sufficient to investigate a case for a single constraint between two support variables. An action a in the domain of a support variable v should be removed in order to establish arc-consistency if it does not have a support with respect to the given constraint. That is all the actions in the domain of the support variable u which neighbors with v through the given constraint are mutex with a. Hence $a \notin dom(u)$ holds. Let us suppose that $\{a\} \cup dom(u)$ is a part of a single action clique of the decomposition. Further let us suppose that action a do not support the atom corresponding to the variable u. Then the projection consistency with respect to a projection goal p which contains the atom corresponding to the variable u removes action a from the action clique. ■

Although situation for the projection consistency from the proof is rather artificial, our empirical experimentation gives us evidence that it is not a rare case. Moreover there are a lot of other situations when projection consistency gives the same results

as arc-consistency. However these situations are difficult to be theoretically classified. Our last note to arc-consistency is that enforcing arc-consistency by the standard *AC-3* algorithm [14] takes $O(|g||A|^3)$ steps for the supports problem consisting of actions A. In contrast the projection consistency requires only $O(|p||A|)$ steps.

Observation 2 (Strength of projection consistency). For a given supports problem S with a goal g there may be a projection goal $p \subseteq g$ such that the problem S is arc-consistent but it is not projection consistent with the projection goal p.

Proof: We will prove the observation by constructing an instance of the supports problem. Let us have a goal $g = \{t_1, t_2, t_3\}$ where t_i for $i = 1,2,3$ are atoms and actions $a_1^1 = (\_, \{t_1\}, \{t_2, t_3\})$, $a_2^1 = (\_, \{t_2\}, \{t_1, t_3\})$, $a_3^1 = (\_, \{t_3\}, \{t_1, t_2\})$, $a_1^2 = (\_, \{t_1\}, \{t_2, t_3\})$, $a_2^2 = (\_, \{t_2\}, \{t_1, t_3\})$ and $a_3^2 = (\_, \{t_3\}, \{t_1, t_2\})$ ($\_$ denotes anonymous variable, that is we do not care about that). It is obvious that the supports problem consisting of actions $\{a_1^1, a_2^1, a_3^1, a_1^2, a_2^2, a_3^2\}$ and the goal g cannot be solved. Actions a_1^1, a_2^1 and a_3^1 are pair-wise mutex as well as actions a_1^2, a_2^2 and a_3^2. The domain of a support variable for the atom t_1 is $\{a_1^1, a_1^2\}$, for the atom t_2 it is $\{a_2^1, a_2^2\}$ and for the atom t_3 it is $\{a_3^1, a_3^2\}$. The arc-consistency does not remove any action from the domains of supports variables. On the other hand, projection consistency is more successful. Suppose that the preprocessing step finds cliques $\{a_1^1, a_2^1, a_3^1\}$ and $\{a_1^2, a_2^2, a_3^2\}$. The contributions of both cliques is 1. Hence no of the actions is supported with respect to projection consistency. So the projection consistency removes all the actions and detects insolvability of the problem. ■

Our preliminary experimentations showed that a good filtering effect can be obtained using projection goals which have the constant number of supports for its atoms. That is the projection consistency is enforced for projection goals $p \subseteq g$ that contains all the atoms for which the number of satisfying actions (see definition 14) is the same. More formally, let $p_i = \{t \mid t \in g \;\&\; |s_t| = i\}$, then projection consistency is enforced for every $i = \{1,2,\ldots\}$ for which $p_i \neq \emptyset$.

The described selection of projection goals prefers situations mentioned within the proofs of observations 1 and 2 (that is, projection consistency with these goals filters more than arc-consistency in most cases). Nevertheless, we do not know whether it is the best set of projection goals with respect to the ratio of pruning power and overall size.

It takes $O(\sum_{i=1,2,\ldots \;\&\; p_i \neq \emptyset} |p_i||A|) = O(|g||A|)$ steps to enforce projection consistency with respect to all projection goals as defined above. If the projection consistency is enforced with respect to one projection goal it may happen that it becomes inconsistent with respect to another projection goal. Therefore the consistency should be enforced repeatedly in *AC-1* style [6] until cliques of actions are no longer changing. This takes $O(|g||A|^2)$ which is still better than $O(|g||A|^3)$ steps of *AC-3*. However empirical tests showed that such repeating does not provide any significant extra filtering effect. Hence we use the only iteration of projection consistency with respect to projection goals p_i for $i = \{1,2,\ldots\}$ where $p_i \neq \emptyset$.

6 Experimental Results

We have implemented the proposed projection consistency propagation algorithm within our experimental planning system written in C++. The projection consistency is used to improve solving of the supports problems within backtracking based plan extraction of the GraphPlan algorithm. We exactly follow the original GraphPlan algorithm except the part for solving the supports problem. The difference is that projection consistency (with respect to projection goals discussed in section 5) is maintained along the search for a solution of the supports problem. Whenever the backtracking algorithm makes a decision (supporting action for an atom is selected) the projection consistency is enforced in order to prune the remaining search space. Our approach is similar to that of Surynek used in [17, 18], but instead of using arc-consistency or singleton arc-consistency we use projection consistency.

We have made several experiments with our algorithm on simple planning domains. We were comparing the standard GraphPlan algorithm and the version of GraphPlan which maintains arc-consistency for solving supports problems with our new version which is maintaining projection consistency.

All the planning problems which were used for our experiments are available at the web site: http://ktiml.mff.cuni.cz/~surynek/research/csclp2007/. The planning domains are the same as that used for empirical tests in [17]. They are Dock Worker Robots planning domain, Refueling Planes planning domain and Towers of Hanoi planning domain. The used planning domains are described in details in [17]. Several problems of varying difficulty of each planning domain were used for our experiments. The planning problems were selected to cover the range from easy problems to relatively hard problems. The lengths of solutions varied from 9 to 38 actions. Performance results on these problems are shown in tables 1, 2 and in figure 1.

Table 1. Time statistics of solving process over several planning problems (part 1) - time is in seconds. The line *Length* shows planning graph length / solution plan length. The line *PlanGraph* shows time spent by building planning graphs, the line *Extraction* shows time spent by extracting plans from planning graphs, the line *Cliques* shows time spent by building clique covers and the line *Total* shows the total time necessary to find one solution.

Problem	han02	pln04	dwr02	dwr01	han04	pln01	han03	pln10	han07
Length	14/14	5/9	6/10	6/12	10/12	5/9	30/30	10/15	14/20
***Standard* GraphPlan**									
PlanGraph	0.90	4.35	5.13	5.23	4.30	15.07	5.47	6.37	14.03
Extraction	0.43	0.28	2.69	8.53	6.75	0.51	12.03	165.82	142.15
Total	1.33	4.63	7.82	13.76	11.05	15.58	17.50	172.19	156.18
GraphPlan with maintaining *arc-consistency* for supports problems									
PlanGraph	0.91	4.11	4.99	4.97	4.25	15.27	5.34	6.37	13.55
Extraction	0.54	0.15	1.59	1.77	3.41	0.36	12.09	36.86	54.57
Total	1.45	4.26	6.58	6.74	7.66	15.63	17.43	43.23	68.12
GraphPlan with maintaining *projection consistency* for supports problems									
PlanGraph	***0.92***	***4.35***	***5.09***	***5.14***	***4.35***	***15.29***	***5.30***	***6.42***	***13.70***
Cliques	***0.06***	***0.33***	***0.16***	***0.15***	***0.24***	***1.18***	***0.24***	***0.45***	***0.86***
Extraction	***0.33***	***0.1***	***0.29***	***0.51***	***1.68***	***0.22***	***5.32***	***9.3***	***21.62***
Total	***1.31***	***4.78***	***5.54***	***5.80***	***6.27***	***16.69***	***10.86***	***16.17***	***36.18***

The tests were performed on a machine with two AMD Opteron 242 processors (2×1600MHz) and 1 GB of memory running Mandriva Linux 10.2. The implementa-

tion was compiled with gcc compiler version 3.4.3 with maximum optimization for the target machine (-O3 -mtune=opteron). No parallel search was used. The two processors were used only for running two tests simultaneously.

The proposed method for solving supports problems based on maintaining of projection consistency brings significant improvement in time of plan extraction as well as in overall problem solving on hard problems compared to the version which uses maintaining of arc-consistency. Notice that the version of the algorithm with projection consistency must perform clique decompositions before the supports problem is solved. Although the clique decompositions represent a slight overhead on easy problems the improvement in plan extraction with projection consistency overrides this disadvantage on hard problems. The improvement of the plan extraction phase is up to about 1000% . Moreover we can say that the larger the portion of time is spent by search the better the improvement by use of projection consistency is.

Table 2. Time statistics of solving process over several planning problems (part 2) - time is in seconds. Several results of the standard GraphPlan are missing due to timeout (2 hours).

Problem	pln05	dwr05	pln06	pln11	dwr07	han08	dwr16	pln13	dwr17
Length	6/14	14/28	9/14	10/14	16/36	20/26	18/34	10/16	20/38
***Standard* GraphPlan**									
PlanGraph	38.6	57.9	44.0	54.8	N/A	39.9	N/A	N/A	N/A
Extraction	460.3	554.3	2660.2	3441.3	N/A	2056.2	N/A	N/A	N/A
Total	499.0	612.2	2704.2	3496.1	N/A	2096.1	N/A	N/A	N/A
GraphPlan with maintaining *arc-consistency* for supports problems									
PlanGraph	38.1	57.2	42.2	53.7	100.50	41.5	207.8	82.3	378.0
Extraction	31.8	60.4	221.2	311.5	279.12	549.8	714.3	1052.5	6148.6
Total	69.9	117.7	263.4	365.2	379.62	591.4	922.2	1134.8	6526.6
GraphPlan with maintaining *projection consistency* for supports problems									
PlanGraph	***40.09***	***57.53***	***44.17***	***56.1***	***99.08***	***40.93***	***204.93***	***86.33***	***369.80***
Cliques	***2.87***	***2.10***	***3.05***	***4.94***	***3.43***	***2.68***	***6.78***	***6.1***	***13.21***
Extraction	***18.92***	***6.13***	***29.45***	***37.16***	***107.74***	***184.02***	***288.61***	***103.81***	***2182.23***
Total	***61.88***	***65.76***	***76.67***	***98.20***	***210.25***	***227.63***	***500.32***	***196.24***	***2565.24***

If we compare the plan extraction which uses projection consistency with the standard version the improvement is up to about 1000% in overall problem solving and up to about 10000% in plan extraction phase. The projection consistency is especially successful on problems with many interacting objects in the planning world and high parallelism of actions (for example refueling planes problems 11 and 13 and dock worker robots problem 05). On the other hand if the interaction between objects in the planning world is low and if there is a low parallelism, the advanced reasoning over supports problems does not represent such formidable improvement (for example refueling planes problem 01 and 05).

Our experimental planning system which we used to produce the empirical tests is not a state-of-the-art planner. At the current stage it cannot compete with planners from International Planning Competition (IPC) [7]. It is caused partially by a not well optimized implementation and partially by the fact that we do not use any domain specific heuristics. Nevertheless it is not our goal to compete with planners participating in IPC at the current stage. We are rather focusing on understanding the structure of planning problems and on utilizing this knowledge to improve the solving process.

For our empirical tests we used standard variable and value selection heuristics. Specifically an atom with the smallest number of supporting actions is always se-

lected as first to be satisfied. Then supporting actions are tried starting with the action that is least constrained. There is also another important implementation issue concerning nogood recording. We use unrestricted nogood recording within our experimental planning system. A special multiple-valued decision tree is used to store nogoods. The tree is optimized for space by preferring low branching towards root and high branching towards leaves of the tree. Our minor experiments showed that the decision tree requires space of about $30\% - 10\%$ of the sum of sizes of all stored nogoods on the testing problems.

The last implementation issue we would like to mention is that we use state variable representation for planning problems [8]. Compared to classical representation the state variable representation provides easier expressing of actions and also some performance advantages directly connected with this fact.

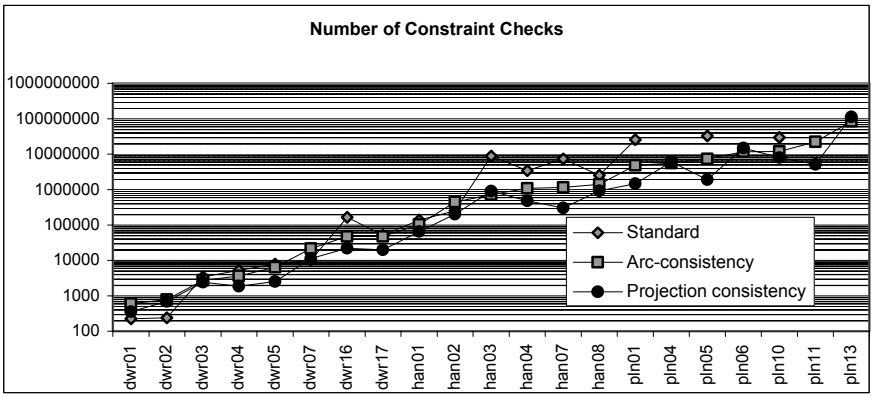


Fig. 1. Number of constraint checks of several planning problems. Several results of the standard GraphPlan are missing due to timeout (2 hours). The checks range uses logarithmic scale.

7 Related Works

The main difference of our approach from other approaches exploiting another formalism (CSP, SAT) for solving planning problems [9, 10, 11, 13] is that we do not formulate the planning problem in another formalism as a whole. We use constraint programming approach only to solve a sub-problem arising during search. Moreover we not only model the supports problems in constraint programming formalism, we extend the formalism by introducing new type of consistency to model the sub-problem in a better way.

Kambhampati's successful idea to formulate plan extraction from planning graph as CSP is presented in [9]. He evaluates the use of various constraint programming techniques and its impact on the effectiveness of plan extraction. Several extensions of expressivity of planning graphs are described in [10]. From our point of view the most interesting idea is to generalize mutex relations and its propagation in planning graph. Another approach is presented in [13] by Lopez and Bacchus. Again they

model the planning problem in planning graph representation as CSP. The originality of their technique consists in making transformations of the obtained CSP which uncovers additional structural information about the problem.

The SAT encodings of planning problems based on planning graphs representation was studied by Kautz and Selman in [11]. The performance of their planner benefits from the performance of SAT solvers. The also successful algorithm CPlan of Van Beek and Chen [20] uses hand tailored CSP encoding of a planning problem. The success of their approach is accounted to well designed numeric constraints that bind spatiotemporally distant object of the planning world.

Finally let us mention that a greedy search for mutex cliques was also used by Blum and Furst in their original GraphPlan [4]. However, they were trying to detect mutex cliques from different reasons. They used the discovered mutex cliques to identify state variables (which we have intrinsically in problem formulation from the beginning) and to reduce memory requirements.

8 Conclusion and Future Work

We proposed a novel consistency technique which we called projection consistency. The technique is designed to prune the search space during extraction of plans by the GraphPlan-style algorithm. We theoretically showed that the projection consistency has faster propagation algorithm than the arc-consistency propagation algorithm AC-3 which application on the same problem was recently studied by Surynek in [17]. Empirical tests showed improvements in order of magnitudes compared to the standard GraphPlan and also compared to the version using arc-consistency. The improvements are both in plan extraction time as well as in overall time.

There is a lot of future work. The first interesting issue is how to make projection consistency stronger. This may be done by other types of projection goals. But it is also possible to do it by slight modification of the definition of the supported action. Instead of the expression $\sum_{j=1,j\neq i}^{n} c(C_j, p) \geq |p - e^+(a)|$ in the definition 12 one can use $\sum_{j=1,j\neq i}^{n} c(C_j, p - e^+(a)) < |p - e^+(a)|$. Unfortunately this change causes that monotonicity (proposition 2) no longer holds. And hence the complexity of propagation algorithm increases. The solution may be a better propagation algorithm.

The similarity between Boolean formula satisfaction problem and supports problem as it is shown in theorem 1 leads us to the question whether it is possible to exploit projection consistency for solving SAT problems. We deal with this question in [19]. The expectable question is also how to extend the presented ideas for planning graphs with time and resources [12, 16]. Since the planning graphs for complex problems are really large the related question is also how to make planning graphs unground and how to get rid of high numbers of no-operation actions.

References

1. Ai-Chang, M., et al.: MAPGEN: Mixed-Initiative Planning and Scheduling for the Mars Exploration Rover Mission. IEEE Intelligent Systems 19(1), 8-12, IEEE Press, 2004.

2. Allen, J., Hendler, J., Tate, A. (editors): Readings in Planning. Morgan Kaufmann Publishers, 1990.
3. Bernard, D. et al.: Remote Agent Experiment. Deep Space 1 Technology Validation Report. NASA Ames and JPL report, 1998.
4. Blum, A. L., Furst, M. L.: Fast Planning through planning graph analysis. Artificial Intelligence 90, 281-300, AAAI Press, 1997.
5. Boeing Co.: Integrated Defense Systems - X-45 J-UCAS. http://www.boeing.com/defense-space/military/x-45/index.html, Boeing Co., USA, October 2006.
6. Dechter, R.: Constraint Processing. Morgan Kaufmann Publishers, 2003.
7. Gerevini, A., Bonet, B., Givan, B. (editors): Fifth International Planning Competition. Event in the context of ICAPS 2006 conference, http://ipc5.ing.unibs.it, University of Brescia, Italy, June 2006.
8. Ghallab, M., Nau, D. S., Traverso, P.: Automated Planning: theory and practice. Morgan Kaufmann Publishers, 2004.
9. Kambhampati, S.: Planning Graph as a (Dynamic) CSP: Exploiting EBL, DDB and other CSP Search Techniques in GraphPlan. Journal of Artificial Intelligence Research 12 (JAIR-12), 1-34, AAAI Press, 2000.
10. Kambhampati, S., Parker, E., Lambrecht, E.: Understanding and Extending GraphPlan. In Proceedings of 4th European Conference on Planning (ECP-97), 260-272, LNCS 1348, Springer-Verlag, 1997.
11. Kautz, H. A., Selman, B.: Unifying SAT-based and Graph-based Planning. In Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI-99), 318-325, Morgan Kaufmann Publishers, 1999.
12. Long, D., Fox, M.: Exploiting a GraphPlan Framework in Temporal Planning. In Proceedings of the 13th International Conference on Automated Planning and Scheduling (ICAPS-2003), 52-61, AAAI Press, 2003.
13. Lopez, A., Bacchus, F.: Generalizing GraphPlan by Formulating Planning as a CSP. In Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI-2003), 954-960, Morgan Kaufmann Publishers, 2003.
14. Mackworth, A. K.: Consistency in Networks of Relations. Artificial Intelligence 8, 99-118, AAAI Press, 1977.
15. Régin, J. C.: A Filtering Algorithm for Constraints of Difference. In Proceedings of the 12th National Conference on Artificial Intelligence (AAAI-94), 362-367, AAAI Press, 1994.
16. Smith, D. E., Weld, D. S.: Temporal Planning with Mutual Exclusion Reasoning. In Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI-99), 326-337, Morgan Kaufmann, 1999.
17. Surynek, P.: Maintaining Arc-consistency over Mutex Relations in Planning Graphs during Search. Accepted to the 20th FLAIRS conference, Key West, Florida, USA, 2007. Technical report no. 328, ITI Series, http://iti.mff.cuni.cz/series/index.html, Charles University, Prague, Czech Republic, 2007.
18. Surynek, P.: Constraint Based Reasoning over Mutex Relations in Planning Graphs during Search. Technical report no. 329, ITI Series, http://iti.mff.cuni.cz/series/index.html, Charles University, Prague, Czech Republic, 2007.
19. Surynek, P.: Solving Difficult SAT Instances Using Greedy Clique Decomposition. Submitted to 7th Symposium on Abstraction, Reformulation, and Approximation (SARA-2007), Canada, 2007. Technical report no. 340, ITI Series, http://iti.mff.cuni.cz/series/index.html, Charles University, Prague, Czech Republic, 2007.
20. Van Beek, P., Chen, X.: CPlan: A Constraint Programming Approach to Planning. In Proceedings of the 16th National Conference on Artificial Intelligence (AAAI-99), 585-590, AAAI Press, 1999.

Breaking Symmetry of Interchangeable Variables and Values

Toby Walsh[1]

National ICT Australia and School of CSE, University of New South Wales, Sydney, Australia, tw@cse.unsw.edu.au

Abstract. A common type of symmetry is when both variables and values partition into interchangeable sets. Polynomial methods have been introduced to eliminate all symmetric solutions introduced by such interchangeability. Unfortunately, whilst eliminating all symmetric solutions is tractable in this case, pruning all symmetric values is NP-hard. We introduce a new propagator for pruning some (but not necessarily all) symmetric values. We show that such static symmetry breaking can be exponentially faster than dynamic methods which eliminate all symmetric branches. This is because the static symmetry breaking constraints may interact with the problem constraints, resulting in addition domain prunings. We also extend such symmetry breaking to interchangeable set variables. Finally, we test these static symmetry breaking constraints experimentally for the first time.

1 Introduction

When solving complex real-life problems like product configuration or staff rostering, symmetry may dramatically increase the size of the search space. A simple and effective mechanism to deal with symmetry is to add static symmetry breaking constraints to eliminate symmetric solutions [1–4]. Alternatively, we can modify the search procedure so that symmetric branches are not explored [5–7]. Unfortunately, eliminating all symmetric solutions is NP-hard in general. In addition, even when all symmetric solutions can be eliminated in polynomial time, pruning all symmetric values may be NP-hard [8]. One way around this problem is to develop polynomial methods for special classes of symmetries.

One common type of symmetry is when variables and/or values are interchangeable. For instance, in a graph colouring problem, if we assign colours (values) to nodes (variables), then the colours (values) are fully interchangeable. That is, we can permute the names of the colours throughout a solution and still have a proper colouring. Similarly, variables may be interchangeable. For example, if two nodes (variables) have the same set of neighbours, we can permute them and keep a proper colouring. We call this "variable and value interchangeability". It has also been called "piecewise symmetry" [9] and "structural symmetry" [10]. Recent results show that we can eliminate all symmetric solutions due to variable and value interchangeability in polynomial time. Sellmann and Van Hententryck give a polynomial time dominance detection algorithm for

dynamically breaking all symmetry introduced by interchangeable variables and values [10]. Subsequently, Flener, Pearson, Sellmann and Van Hentenryck identified a set of static symmetry breaking constraints to eliminate all symmetric solutions [9]. In this paper, we study such symmetry breaking in more detail.

2 Background

A constraint satisfaction problem (CSP) consists of a set of n variables, each with a domain of values, and a set of constraints specifying allowed combinations of values for given subsets of variables. A constraint restricts values taken by some subset of variables to a subset of the Cartesian product of their domains. Without loss of generality, we assume that variables initially share the same domain of possible values, d_1 to d_m. Finite domain variables take one value from this domain. Set variables take sets of such values and are typically defined by a lower bound on the definite elements and an upper bound on the definite and potential elements. We also assume an ordering on values in which $d_i < d_j$ iff $i < j$. A solution is an assignment of values to variables satisfying the constraints.

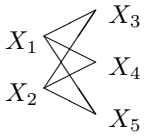
A global constraint involves a parameterised number of variables. We will use three common global constraints. The first, $\textsc{Among}([X_1, .., X_n], v, M)$ holds iff $|\{i \mid X_i \in v\}| = M$. That is, M of the variables, X_1 to X_n take values among the set v. Combining together multiple Among constraints gives the global cardinality constraint. $\textsc{Gcc}([X_1, .., X_n], [d_1, .., d_m], [O_1, .., O_m])$ holds iff $|\{i \mid X_i = d_j\}| = O_j$ for $1 \leq j \leq m$. That is, O_j of the variables, X_1 to X_n take the value d_j. Finally, a global constraint that we will use to encode other global constraints is the Regular constraint. This ensures that the values taken by a sequence of variables form a string accepted by a finite automaton [11]. Quimper and Walsh encode a linear time GAC propagator for the Regular constraint using simple ternary constraints [12]. They introduce variables for the state of the automaton after each character has been read, and post ternary constraints ensuring that the state changes according to the transition relation. One advantage of this encoding is that we have easy access to the states of the automaton. In fact, we will need here to link the final state to a finite domain variable.

Systematic constraint solvers typically explore partial assignments using backtracking search, enforcing a local consistency to prune values for variables which cannot be in any solution. We consider two well known local consistencies: generalized arc consistency and bound consistency. Given a constraint C on finite domain variables, a *support* is assignment to each variable of a value in its domain which satisfies C. A constraint C on finite domains variables is *generalized arc consistent* (GAC) iff for each variable, every value in its domain belongs to a support. Given a constraint C on set variables, a *bound support* on C is an assignment of a set to each set variable between its lower and upper bounds which satisfies C. A constraint C is *bound consistent* (BC) iff for each set variable S, the values in $ub(S)$ belong to S in at least one bound support and the values in $lb(S)$ belong to S in all bound supports.

3 Variable and value interchangeability

We suppose that there is a partition of the n finite domain variables of our CSP into a disjoint sets, and the variables within each set are interchangeable. That is, if we have a solution, $\{X_i = d_{sol(i)} \mid 1 \leq i \leq n\}$ and any bijection σ on the variable indices which permutes indices within each partition, then $\{X_{\sigma(i)} = d_{sol(i)} \mid 1 \leq i \leq n\}$ is also a solution. We also suppose that there is a partition of the m values into b disjoint sets, and the values within each set are interchangeable. That is, if we have a solution, $\{X_i = d_{sol(i)} \mid 1 \leq i \leq n\}$ and any bijection σ on the value indices which permutes indices within each partition, then $\{X_i = d_{\sigma(sol(i))} \mid 1 \leq i \leq n\}$ is also a solution. If $n = a$ then we have just interchangeable values, whilst if $m = b$ we have just interchangeable variables. We will order variable indices so that $X_{p(i)}$ to $X_{p(i+1)-1}$ is the ith partition of variables, and value indices so that $d_{q(j)}$ to $d_{q(j+1)-1}$ is the jth partition of values where $1 \leq i \leq a$, $1 \leq j \leq b$.

Example 1 *Consider a CSP problem representing 3-colouring the following graph:*



Nodes are labelled with the variables X_1 to X_5. Values correspond to colours. X_1 and X_2 are interchangeable as the corresponding nodes have the same set of neighbours. If we have a proper colouring, we can permute the values assigned to X_1 and X_2 and still have a proper colouring. Similarly, X_3, X_4 and X_5 are interchangeable. The variables thus partition into two interchangeable sets: $\{X_1, X_2\}$ and $\{X_3, X_4, X_5\}$. In addition, we can uniformly permute the colours throughout a solution and still have a proper colouring. Thus, the values partition into a single interchangeable set: $\{d_1, d_2, d_3\}$.

Flener *et al.* [9] show that we can eliminate all solutions which are symmetric due to variable and value interchangeability by posting the following constraints:

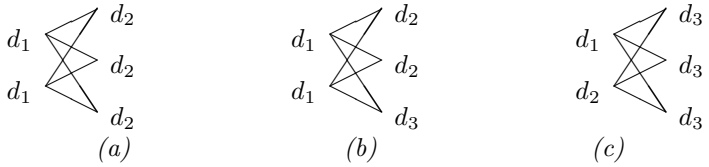
$$X_{p(i)} \leq .. \leq X_{p(i+1)-1} \qquad \forall\ i \in [1, a] \qquad (1)$$

$$\textsc{Gcc}([X_{p(i)}, .., X_{p(i+1)-1}], [d_1, .., d_m], [O^i_1, .., O^i_m]) \qquad \forall\ i \in [1, a] \qquad (2)$$

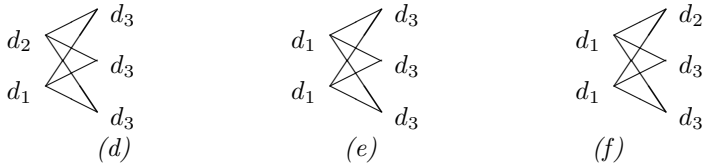
$$(O^1_{q(j)}, .., O^a_{q(j)}) \geq_{\text{lex}} .. \geq_{\text{lex}} (O^1_{q(j+1)-1}, .., O^a_{q(j+1)-1}) \qquad \forall\ j \in [1, b] \qquad (3)$$

$(O^1_k, .., O^a_k)$ is the so called *signature* of the value d_k. The signature gives the number of occurrences of the value d_k in each equivalence class of variables. This identifies apart the values. Note that the signature is invariant to the permutation of variables within each equivalence class. By ordering variables within each equivalence class using (1), we rule out permuting interchangeable variables. Similarly, by ordering the signatures of values within each equivalence class using (3), we rule out permuting interchangeable values.

Example 2 *Consider again the 3-colouring problem in Example 1. There are 30 proper colourings of this graph. When we post the above symmetry breaking constraints, the number of proper colourings reduces from 30 to just 3:*



Each colouring is representative of a different equivalence class. In fact, it is the lexicographically least member of its equivalence class. On the other hand, the following colourings are eliminated by the above symmetry breaking constraints:



For instance, the proper colouring given in (e) is symmetric to that given in (a) since if we permute d_2 with d_3 in (e), we get (a). The proper colouring given in (e) is eliminated by the symmetry breaking constraint $(O_2^1, O_2^2) \geq_{\text{lex}} (O_3^1, O_3^2)$ since $O_2^1 = O_3^1 = 0$ (neither d_2 nor d_3 occur in the first equivalence class of variables) but $O_2^2 = 0$ and $O_3^2 = 3$ (d_2 does not occur in the second equivalence class of variables but d_3 occurs three times).

Suppose BreakInterchangeability$(p, q, [X_1, .., X_n])$ is a global constraint that eliminates all symmetric solutions introduced by interchangeable variables and values. That is, BreakInterchangeability orders the variables within each equivalence class, as well as lexicographical ordering the signatures of values within each equivalence class. It can be seen as the conjunction of the ordering, Gcc, and lexicographical ordering constraints given in Equations (1), (2) and (3). Enforcing GAC on such a global constraint will prune all symmetric values due to variable and value interchangeability. Not surprisingly, decomposing this global constraint into separate ordering, Gcc and lexicographical ordering constraints may hinder propagation.

Example 3 *Consider again the 3-colouring problem in Example 1. Suppose X_1 to X_5 have domains $\{d_1, d_2, d_3\}$, the signature variables O_1^1, O_2^1, O_3^1 have domains $\{0, 1, 2\}$, whilst O_1^2, O_2^2, O_3^2 have domains $\{0, 1, 2, 3\}$. Flener et al.'s decomposition and the binary not-equals constraints between variables representing neighbouring nodes are GAC. However, by considering (a), (b) and (c), we see that enforcing GAC on* BreakInterchangeability *and the binary not-equals constraints ensures $X_1 = d_1$, $X_2 \neq d_3$, $X_3 \neq d_1$, $X_4 \neq d_1$ and $X_5 \neq d_1$.*

As decomposing BreakInterchangeability hinders propagation, we might consider a specialised propagation algorithm for achieving GAC that prunes all

possible symmetric values. Unfortunately enforcing GAC on such a global constraint is NP-hard [8]. We cannot therefore expect to find a polynomial time propagation algorithm to prune all symmetric values (assuming $P \neq NP$).

4 A new decomposition

We propose an alternative decomposition of BreakInterchangeability. This decomposition does not need global cardinality constraints which are expensive to propagate. In fact, Flener *et al.*'s decomposition requires a propagator for Gcc which prunes the bounds on the number of occurrence of values. This is not available in several solvers including Sicstus and Eclipse. By comparison, the decomposition proposed here uses just Regular constraints which are available in many solvers or can be easily added using simple ternary transition constraints [12]. This new decomposition can be efficiently and incrementally propagated.

The results in Table 5 of [13] suggest that propagation is rarely hindered by decomposing a chain of lexicographical ordering constraints into individual lexicographical ordering constraints between neighbouring vectors. Results in Table 1 [14] also suggest that propagation is rarely hindered by decomposing symmetry breaking constraints for interchangeable values into symmetry breaking constraints between neighbouring pairs of values in each equivalence class. We therefore propose a decomposition which only considers the signatures of neighbouring pairs of values in each equivalence class.

This decomposition replaces BreakInterchangeability by a linear number of symmetry breaking constraints, SigLex. These lexicographically order the signatures of neighbouring pairs of values in each equivalence class, as well as ordering variables within each equivalence class. More precisely, we introduce SigLex$(k, [X_1, .., X_n])$ where $q(j) \leq k < q(j+1) - 1,\ 1 \leq j \leq b$. The global constraint SigLex$(k, [X_1, .., X_n])$ itself holds iff:

$$X_{p(i)} \leq .. \leq X_{p(i+1)-1} \qquad \forall\ i \in [1, a] \tag{4}$$

$$\text{Among}([X_{p(i)}, .., X_{p(i+1)-1}], \{d_k\}, O_k^i) \qquad \forall\ i \in [1, a] \tag{5}$$

$$\text{Among}([X_{p(i)}, .., X_{p(i+1)-1}], \{d_{k+1}\}, O_{k+1}^i) \quad \forall\ i \in [1, a] \tag{6}$$

$$(O_k^1, .., O_k^a) \geq_{\text{lex}} (O_{k+1}^1, .., O_{k+1}^a) \tag{7}$$

SigLex orders the variables within each equivalence class and lexicographically orders the signature of two values which are interchangeable and neighbouring to each other. To propagate each SigLex constraint, we give a decomposition using Regular constraints which does not hinder propagation.

Theorem 1 *GAC can be enforced on* SigLex$(k, [X_1, .., X_n])$ *in* $O(n^2)$ *time.*

Proof: We first post ordering constraints, $X_{p(i)} \leq .. \leq X_{p(i+1)-1}$ on each equivalence class of variables. We then channel into a sequence of four valued variables using the constraints: $Y_i^k = (X_i > d_{k+1}) + (X_i \geq d_{k+1}) + (X_i \geq d_k)$. That is, if $X_i > d_{k+1}$ then $Y_i^k = 3$, else if $X_i = d_{k+1}$ then $Y_i^k = 2$, else if $X_i = d_k$ then $Y_i^k = 1$, else $X_i < d_k$ and $Y_i^k = 0$.

Within the ith equivalence class of variables, we post a REGULAR constraint on $Y^k_{p(i)}$ to $Y^k_{p(i+1)-1}$ to compute the difference between O^i_k and O^i_{k+1} and assign this difference to a new integer variable D^i_k. The automaton associated with this REGULAR constraint has state variables $Q^k_{p(i)}$ to $Q^k_{p(i+1)-1}$ whose values are tuples containing the difference between the two counts seen so far as well as the last value seen (so that we can ensure that values for Y^k_i are increasing). From (δ, y), the transition function on seeing Y^k_i moves to the new state $(\delta + (Y^k_i = 2) - (Y^k_i = 1), \max(y, Y^k_i))$ if and only if $Y^k_i \geq y$. The initial state is $(0, 0)$. We set the difference between the two counts in the final state variable equal to the new integer variable D^i_k (which is thus constrained to equal $O^i_{k+1} - O^i_k$) Finally, we ensure that the vectors, $(O^1_k, .., O^a_k)$ and $(O^1_{k+1}, .., O^a_{k+1})$ are ordered using a final REGULAR constraint on the difference variables, D^1_k to D^a_k. The associated automaton has 0/1 states, a transition function which moves from state b to $b \vee (D^i_k < 0)$ provided $D^i_k \leq 0$ or $b = 1$, an initial state 0 and 0 or 1 as final states.

The constraint graph of the ternary decompositions of all the REGULAR constraints is Berge-acyclic. Hence enforcing GAC on these ternary constraints achieves GAC on the variables Y^k_i. Consider a support for the Y^k_i variables. We can extend this to a support for the X_i variables simply by picking the smallest value left in their domains after we have enforced GAC on the channelling constraints between the X_i and Y^k_i variables. Support for values left in the domains of the X_i variables can be constructed in a similar way. Enforcing GAC on this decomposition therefore achieves GAC on SIGLEX$(k, [X_1, .., X_n])$.

Enforcing GAC on the ordering constraints takes $O(n)$ time (assuming bounds can be accessed and updated in constant time), on the channelling constraints between X_i and Y^k_i takes $O(n)$ time (again assuming bounds can be accessed and updated in constant time), on the first set of REGULAR constraints which compute D^i_k takes $O(n^2)$ time, and on the final REGULAR constraint takes $O(na)$ time. As $a \leq n$, enforcing GAC on SIGLEX takes $O(n^2)$ time. $\diamond$

We compare this with the GCC decomposition given in [9]. This requires a propagator for GCC which prunes the bounds of the occurrence variables. This will take $O(mn^2 + n^{2.66})$ time [15]. To break the same set of symmetries, we need to post up to $O(m)$ SIGLEX constraints, which take $O(mn^2)$ time in total to propagate. In the best case for this new decomposition, m grows slower that $O(n^{0.66})$ and we are faster. In the worst case, m grows as $O(n^{0.66})$ or worse and both propagators take $O(mn^2)$ time. The new decomposition is thus sometimes better but not worse than the old one. The amount of pruning achieved using the two decompositions is incomparable in general.

Theorem 2 *GAC on a set of* SIGLEX *constraints is incomparable to GAC on the equivalent* GCC *decomposition.*

Proof: Suppose all variables and values are interchangeable.

Consider $X_1 = d_1$, and X_2, X_3 and $X_4 \in \{d_2, d_3\}$. Then SIGLEX$(1, [X_1, \ldots, X_4])$, and SIGLEX$(2, [X_1, \ldots, X_4])$ are GAC. However, enforcing GAC on the GCC decomposition causes a domain wipe-out as d_2 or d_3 must occur more than d_1.

Consider X_1 and $X_2 \in \{d_1, d_2\}$. Then enforcing GAC on the Gcc decomposition does not prune the domain of X_1. However, enforcing GAC on SigLex$(1, [X_1, X_2])$ prunes d_2 from the domain of X_1. $\diamond$

Whilst the two decompositions are incomparable, we can exhibit a problem on which the new decomposition gives exponential savings. We conjecture that the reverse is also true.

Theorem 3 *On the pigeonhole problem, $PHP(n)$ with n interchangeable variables and $n+1$ interchangeable values, we explore $O(2^n)$ branches when maintaining GAC and breaking symmetry using the* Gcc *decomposition irrespective of the variable and value ordering, but we solve in polynomial time when enforcing GAC using* SigLex *constraints.*

Proof: The problem has $n+1$ constraints of the form $\bigvee_{i=1}^{n} X_i = d_j$ for $1 \leq j \leq n+1$, with $X_i \in \{d_1, .., d_{n+1}\}$ for $1 \leq i \leq n$. The problem is unsatisfiable by a simple pigeonhole argument. Enforcing GAC on SigLex$(i, [X_1, \ldots, X_n])$ for $i > 0$ prunes d_{i+1} from X_1. Hence, X_1 is set to d_1. Enforcing GAC on SigLex$(i, [X_1, \ldots, X_n])$ for $i > 1$ now prunes d_{i+1} from X_2. The domain of X_2 is thus reduced to $\{d_1, d_2\}$. By a similar argument, the domain of each X_i is reduced to $\{d_1, \ldots d_i\}$. The SigLex constraints are now GAC. Enforcing GAC on the constraint $\bigvee_{i=1}^{n} X_i = d_{n+1}$ then proves unsatisfiability. Thus, we prove that the problem is unsatisfiable in polynomial time. On the other hand, using the Gcc decomposition, irrespective of the variable and value ordering, we will only terminate each branch when $n-1$ variables have been assigned (and the last variable is forced). A simple calculation shows that the size of the search tree as least doubles as we increase n by 1. Hence we will visit $O(2^n)$ branches before declaring the problem unsatisfiable. $\diamond$

5 Some special cases

Variables are not interchangeable

Suppose we have interchangeable values but no variable symmetries (i.e. $a = n$ and $b < m$). To eliminate all symmetric solutions in such a situation, Law and Lee introduced value precedence [4]. This breaks symmetry by constraining when a value is first used. More precisely, Precedence$(k, [X_1, .., X_n])$ holds iff $\min\{i \mid X_i = d_k \vee i = n+1\} < \min\{i \mid X_i = d_{k+1} \vee i = n+2\}$. That is, the first time we use d_k is before the first time we use d_{k+1}. This prevents the two values being interchanged. It is not hard to show that the SigLex constraint is equivalent to value precedence in this situation.

Theorem 4 Precedence$(k, [X_1, .., X_n])$ *is equivalent to* SigLex$(k, [X_1, .., X_n])$ *when $n = a$.*

Proof: If $n = a$ then the vectors computed within SigLex, $(O_k^1, .., O_k^a)$ and $(O_{k+1}^1, .., O_{k+1}^a)$, are n-ary 0/1 vectors representing the indices at which d_k and

d_{k+1} appear. Lexicographically ordering these vectors ensures that either d_k is used before d_{k+1} or neither are used. This is equivalent to value precedence. $\diamond$

In this case, the propagator for SigLex mirrors the work done by the propagator for Precedence given in [14]. Although the two propagators have the same asymptotic cost, we might prefer the propagator for Precedence as it introduces fewer intermediate variables.

All variables and values are interchangeable

Another special case is when all variables and values are fully interchangeable (i.e. $a = b = 1$). To eliminate all symmetric solutions in such a situation, Walsh introduced a global constraint which ensures that the sequence of values is increasing but the number of their occurrences is decreasing [14]. More precisely, DecSeq($[X_1, .., X_n]$) holds iff $X_1 = d_1$, $X_i = X_{i+1}$ or ($X_i = d_j$ and $X_{i+1} = d_{j+1}$) for $1 \leq i < n$ and $|\{i \mid X_i = d_k\}| \geq |\{i \mid X_i = d_{k+1}\}|$ for $1 \leq k < m$. Not surprisingly, the SigLex constraint ensures such an ordering of values.

Theorem 5 *If $a = b = 1$ then* SigLex($k, [X_1, .., X_n]$) *for $1 \leq k < m$ is equivalent to* DecSeq($[X_1, .., X_n]$).

Proof: Suppose SigLex($k, [X_1, .., X_n]$) holds for $1 \leq k < m$. Then $O_k^1 \geq O_{k+1}^1$ for $1 \leq k < m$. Now $O_k^1 = |\{i \mid X_i = d_k\}|$. Hence $|\{i \mid X_i = d_k\}| \geq |\{i \mid X_i = d_{k+1}\}|$ for $1 \leq k < m$. Suppose $O_1^1 = 0$. Then $O_k^1 = 0$ for $1 \leq k \leq m$ and no values can be used. This is impossible. Hence $O_1^1 > 0$ and d_1 is used. As $X_1 \leq .. \leq X_n$, $X_1 = d_1$. Suppose that d_k is the first value not used. Then $O_k^1 = 0$. Hence $O_j^1 = 0$ for all $j > k$. That is, all values up to d_k are used and all values including and after d_k are not used. Since $X_i \leq X_{i+1}$, it follows that $X_i = X_{i+1}$ or ($X_i = d_j$ and $X_{i+1} = d_{j+1}$) for $1 \leq i < n$. Thus, DecSeq($[X_1, .., X_n]$) holds. The proof reverses easily. $\diamond$

6 Dynamic methods

Sellmann and Van Hententryck give a polynomial time dominance detection algorithm for interchangeable variables and values [10]. Such a dominance detection algorithm can be incorporated into a dynamic symmetry breaking method like the SBDD method so that symmetric branches are never explored. Such dynamic methods may be less powerful than static symmetry breaking constraints. The reason is that dynamic methods only prune the domain of the current variable. With static symmetry breaking constraints, we may also prune any *future* variables. Such prunings may then cause the problem constraints to propagate. This interaction between the original constraints of the problem and the added symmetry breaking constraints can lead to significant reductions in the size of the search tree.

Theorem 6 *On the pigeonhole problem, $PHP(m)$ where $m = n^2+1$ we explore $O(2^n)$ branches when maintaining GAC and breaking symmetry using SBDD, but we solve in polynomial time when enforcing GAC using* SigLex *constraints.*

Proof: Irrespective of the variable and value ordering, SBDD will only terminate each branch when $m-1$ variables have been assigned (and the last variable is forced). Thus SBDD will explore all undominated $m-1$ variable assignments. Each such undominated assignment can be described by its signature. The signature is an ordered tuple of integers which add up to $m-1$. The total number of such signatures is simply the total number of partitions of $m-1$, $P(m-1)$. Hardy and Ramanujan (1918) showed that $P(m)$ converges to $\frac{1}{4m\sqrt{3}}e^{\pi\sqrt{2m/3}}$. Using some basic algebraic manipulation with $m = n^2+1$, we calculate that there are $O(2^n)$ partitions of $m-1$. Thus, SBDD will explore an exponential number of branches. By comparison, enforcing GAC on SigLex constraints prunes d_{m+1}. The corresponding disjunction is thus shown unsatisfiable without backtracking. ◇

Note that, by Theorem 3, we also take exponential time on this problem using the Gcc decomposition. Note that this does not contradict Theorem 2 in [9]. This shows that dynamic and static methods for breaking the symmetry of interchangeable variables and values explore the same search tree. However, this result is in the absence of any problem constraints. It is precisely the interaction between the symmetry breaking constraints and the problem constraints that give static methods a potential advantage over dynamic methods.

7 Experimental results

As the Gcc and SigLex decompositions are incomparable, we tested them experimentally. We coded all problems with the finite domain library in BProlog and ran them on a PowerPC 1GHz G4 processor with 1.25 GB RAM. This is the first time that these symmetry breaking methods have been tested empirically.

7.1 Pigeonhole problems

We first tried problems in which all variables and values are interchangeable (i.e. $a = b = 1$). We used the pigeonhole problems, $PHP(n)$ mentioned earlier. Results are given in Table 1. As predicted by Theorem 3, the new decomposition using SigLex constraints performs well on such problems.

7.2 Schur numbers

We next tried problems in which all values but none of the variables are interchangeable (i.e. $a = n$ and $b = 1$). We used the Schur number problem (prob015 in CSPLib). This has been used in several previous experimental studies of symmetry breaking [4, 14]. The Schur number $S(k)$ is the largest integer n such that the interval $[1, n]$ can be partitioned into k sum-free sets. S is sum-free iff

$PHP(n)$	symmetry breaking					
	none		Gcc decomposition		SigLex decomposition	
	b	**t**	**b**	**t**	**b**	**t**
4	124	**0.00**	44	**0.00**	**0**	**0.00**
5	1,295	0.02	265	0.01	**0**	**0.00**
6	16,806	0.23	1,722	0.10	**0**	**0.00**
7	262,143	3.88	13,545	0.87	**0**	**0.00**
8	4,782,968	78.11	114,208	8.63	**0**	**0.00**
9			1,099,314	95.63	**0**	**0.00**
10					**0**	**0.00**

Table 1. Pigeonhole problems: **b**acktracks and **t**ime to solve in secs using a fail first heuristic. Blank entries are problems not solved in 10 minutes.

$S(n,k)$	symmetry breaking					
	none		Gcc decomposition		SigLex decomposition	
	b	**t**	**b**	**t**	**b**	**t**
$S(13,3)$	173	**0.01**	31	**0.01**	**28**	**0.01**
$S(13,4)$	1,192,535	11.61	49,793	1.72	**49,198**	**1.36**
$S(13,5)$			692,567	24.45	**685,463**	**20.90**
$S(13,6)$			2,551,207	101.20	**2,473,321**	**85.59**
$S(14,3)$	161	0.01	29	0.01	**26**	**0.01**
$S(14,4)$	2,335,799	25.89	97,457	3.60	**95,311**	**2.16**
$S(14,5)$			2,149,785	61.96	**2,127,353**	**53.21**
$S(14,6)$			10,644,774	442.05	**10,384,555**	**337.04**
$S(15,3)$	161	**0.01**	29	**0.01**	**26**	**0.01**
$S(15,4)$	6,021,071	50.93	250,879	7.08	**248,437**	**5.59**
$S(15,5)$			8,278,307	239.97	**8,229,688**	**202.25**
$S(15,6)$						

Table 2. Schur numbers problem: **b**ranches and **t**ime to find all solutions in secs using a fail first heuristic. Blank entries are problems not solved in 10 minutes.

$\forall a, b, c \in S \; . \; a \neq b + c$. We consider the corresponding decision problem, $S(n, k)$ which asks if the interval $[1, n]$ can be partitioned into k sum-free sets. A simple model of this uses n finite domain variables with k interchangeable values. Results are given in Table 2. We explore slightly fewer branches using the SigLex decomposition, and this model is roughly 20% faster in cpu time.

7.3 Template design

We then ran experiments in which we have interchangeable values and variables, but the number of equivalence classes is small (i.e. a and b are large). We used the template design problem (prob002 in CSPLib). We took a simple model with a variable for each slot on a template, whose value is the design printed here. This model has interchangeable variables (as slots within a template can be permuted) and interchangeable values (as designs with the same demand can be permuted). As in previous studies, we consider a decision version of the problem where we limit over production of any design to p%. Results are given in Table 3 for the

cat food problem. To start the search for a design, we need an upper bound on the run length for any template. In these experiments, we limited production of any template to $600,000/k$ which gave us feasible solutions wherever they were possible. Without symmetry breaking, we were unable to solve any of the problems. Both decompositions appear to be equally effective at dealing with this type of symmetry. However, the GCC decomposition (which uses a built-in GCC propagator as opposed to our hand crafted REGULAR constraint) is a few percent faster in cpu time.

templates k	over production $p\%$	symmetry breaking GCC decomposition **b**	**t**	SIGLEX decomposition **b**	**t**
2	10%	**1,770**	**0.98**	**1,770**	1.01
3		**12,614**	**10.03**	**12,614**	10.25
4		**284,659**	**267.19**	**284,659**	274.55
2	5%	**960**	**0.54**	**960**	0.56
3		**26,999**	**21.66**	**26,999**	21.90
4		**225,444**	**211.25**	**225,444**	215.40
2	2.5%	**333**	**0.23**	**333**	**0.23**
3		**45,895**	**36.21**	**45,895**	36.70
4		**266,153**	**248.86**	**266,153**	256.43

Table 3. Cat food template design problem: **b**acktracks, and **t**ime to find a solution in secs using a fail first heuristic.

7.4 *n* by *n* queens

We end with experiments testing the interaction with other types of symmetry breaking constraints. We used the n by n queens problem which appears in other studies of symmetry breaking [8, 16]. The aim is to colour the squares in a n by n chessboard with one of n colours so that no row, column or diagonal has the same colour twice. We model this with n^2 finite domain variables, each with n possible values, and an all different constraint along each row, column and diagonal. The model has 8 variable symmetries corresponding to the rotations and reflections of the chessboard. We break these symmetries by posting suitable ordering constraints (for example, that the top left is coloured less than the top right). The model also has value symmetry as all colours are interchangeable. We break these with either the GCC or SIGLEX decompositions. Results are given in Table 4. We explore slightly fewer branches using the SIGLEX decomposition, and this model is again approximately 20% faster in cpu time.

7.5 Summary of experimental results

The SIGLEX decomposition appears to be effective at breaking symmetry, especially when we have many interchangeable values and/or variables. On the

n	symmetry breaking					
	none		GCC decomposition		SIGLEX decomposition	
	b	**t**	**b**	**t**	**b**	**t**
4	6	**0.00**	4	**0.00**	**0**	**0.00**
5	58	**0.00**	9	0.01	**1**	**0.00**
6	3948	0.09	42	0.03	**29**	**0.01**
7	882,812	22.04	858	0.44	**839**	**0.32**
8			148,589	81.21	**148,563**	**65.57**
9						

Table 4. n by n queens problem: **b**acktracks, and **t**ime to find all solutions in secs using a fail first heuristic. Blank entries are problems not solved in 10 minutes.

range of problems tested here, it was either comparable to or better than the GCC decomposition.

8 Interchangeable set variables with interchangeable values

Interchangeable variables and values can also occur in problems containing set variables. We suppose there is a partition of the n set variables in a problem into a disjoint sets where the variables within each set are fully interchangeable, and a partition of the m values taken by these set variables into b disjoint sets where the values within each set are also fully interchangeable. For example, in one model of the social golfers problem (prob010 in CSPLib), the set variables representing groups can be partitioned into weeks (as groups playing in a given week can be permuted), whilst the values are fully interchangeable (as we can freely permute the names of the golfers). We again order the set variables so that $S_{p(i)}$ to $S_{p(i+1)-1}$ is the ith partition of variables, and the values so that $d_{q(j)}$ to $d_{q(j+1)-1}$ is the jth partition of values.

We can lift the SIGLEX constraint to set variables in a straight forward way. Given a sequence of interchangeable set variables, we let the *signature* of the value d_k be the number of occurrences of this value within each equivalence class of set variables. More precisely, the signature is the vector $(O_k^1, .., O_k^a)$ where $O_k^i = |\{j \mid d_k \in S_j, p(i) \leq j < p(i+1)\}|$. The global constraint $\text{SIGLEX}_{\text{set}}$ lexicographically orders the signatures of two neighbouring and interchangeable values as well as ordering the set variables within each equivalence class. More precisely, $\text{SIGLEX}_{\text{set}}(k, [S_1, .., S_n])$ holds iff $(O_k^1, .., O_k^a) \geq_{\text{lex}} (O_{k+1}^1, .., O_{k+1}^a)$ and $S_{p(i)} \leq_{\text{mset}} .. \leq_{\text{mset}} S_{p(i+1)-1}$ for $1 \leq i \leq a$. The multiset ordering, $\leq_{\text{mset}}$ on set variables is equivalent to the lexicographical ordering on the 0/1 vector representing their characteristic function. We post $\text{SIGLEX}_{\text{set}}(k, [S_1, .., S_n])$ for $q(j) \leq k < q(j+1)$ and $1 \leq j \leq b$. This can again be propagated using a decomposition based on REGULAR constraints.

We say that a set of symmetry breaking constraints is *consistent* iff for each equivalence class of assignments, they leave at least one symmetric assignment.

We say that a set of symmetry breaking constraints is *complete* iff for each equivalence class of assignments, they leave at most one symmetric assignment. Whilst posting SigLex constraints is both consistent and complete [9], SigLex$_{\text{set}}$ constraints are consistent but not complete.

Theorem 7 SigLex$_{\text{set}}$ *constraints are consistent but not complete for breaking the symmetry of interchangeable set variables and values.*

Proof: Lexicographically ordering the signatures will pick out one or more assignments within each equivalence class. As the set variables within each equivalence class are interchangeable, we can order them using any total ordering like the multiset ordering. This leaves the signature of each value unchanged. Hence, we can first order the signatures of interchangeable values and then order the set variables within a partition without eliminating all assignments within each equivalence class. Thus, it is consistent to post SigLex$_{\text{set}}$ constraints. To show that we may not eliminate all symmetric solutions, consider $a = b = 1$, $n = 2$ and $m = 3$. Suppose $S_1 = \{d_1\}$ and $S_2 = \{d_2, d_3\}$. Then SigLex$_{\text{set}}(1, [S_1, S_2])$ and SigLex$_{\text{set}}(2, [S_1, S_2])$ hold. Now if we interchange d_1 and d_3 and S_1 and S_2, we get $S_1 = \{d_1, d_2\}$ and $S_2 = \{d_3\}$ which also satisfies SigLex$_{\text{set}}(1, [S_1, S_2])$ and SigLex$_{\text{set}}(2, [S_1, S_2])$. $\diamond$

The fact that SigLex$_{\text{set}}$ constraints may not be complete should perhaps not be too surprising as eliminating all symmetric solutions is NP-hard in this case [10]. However, we can identify a special case where SigLex$_{\text{set}}$ constraints are complete and all symmetry can be broken in polynomial time. Suppose $a = n$ and we have set variables that are not symmetric, but interchangeable values. To deal with this situation, Law and Lee introduced the global precedence constraint over set variables [4]. More precisely, Precedence$_{\text{set}}(k, [S_1, .., S_n])$ holds iff $\min\{i \mid (d_k \in S_i \wedge d_{k+1} \notin S_i) \vee i = n+1\} < \min\{i \mid (d_{k+1} \in S_i \wedge d_k \notin S_i) \vee i = n+2\}$. That is, if we distinguish apart d_k from d_{k+1} (by one occurring in a set without the other) then d_k occurs first without d_{k+1}. Posting Precedence$_{\text{set}}$ constraints for each pair of interchangeable values is a consistent and complete method to break the symmetry of interchangeable values (Theorem 4.1 in [17]). In this case, the SigLex$_{\text{set}}$ constraints ensure such value precedence and are thus consistent and complete.

Theorem 8 *When* $a = n$, SigLex$_{\text{set}}(k, [S_1, .., S_n])$ *is equivalent to* Precedence$_{\text{set}}(k, [S_1, .., S_n])$

Proof: If $a = n$ then the signature $(O_k^1, .., O_k^a)$ is just a n-ary 0/1 vector indicating whether the value d_k occurs in each set variable. Lexicographically ordering these vectors ensures that either d_k occurs on its own before d_{k+1} or they both occur or not occur together. This is equivalent to value precedence. $\diamond$

9 Related work

Puget proved that symmetries can always be eliminated by the additional of suitable constraints [1]. Crawford *et al.* presented the first general method for

constructing such symmetry breaking constraints [2]. We add so-called "lex-leader" constraints which ensure that the solution is lexicographically less than any of its symmetries. Crawford *et al.* also argued that it is NP-hard to eliminate all symmetric solutions in general.

The full set of lex-leader constraints can often be simplified. For example, if we have an array of decision variables with row symmetry (that is, the rows can be permuted), the exponential number of lex-leader constraints simplifies to a linear number of lexicographical ordering constraints between rows [18, 3]. As a second example, for problems where variables are symmetric and must take all different values, Puget has shown that the lex-leader constraints simplify to a linear number of binary inequality constraints [19].

To break value symmetry, Puget introduces one variable per value and a linear number of binary ordering constraints [16]. To deal with the special type of value symmetry where values are interchangeable, Law and Lee formally defined value precedence and proposed a specialised propagator for breaking the symmetry of a pair of interchangeable values [4]. Walsh extended this to a propagator for any number of interchangeable values [14]. Finally, an alternative way to break value symmetry statically is to convert it into a variable symmetry by channelling into a dual viewpoint and using lexicographical ordering constraints on this dual view [3, 20].

10 Conclusions

We have considered breaking the symmetry introduced by interchangeable variables and values. Whilst there exist polynomial methods to eliminate all symmetric solutions introduced by interchangeable variables and values, pruning all symmetric values is NP-hard. We have introduced a new propagator for pruning some (but not necessarily all) symmetric values. The new propagator is based on a decomposition using Regular constraints. We have shown that a backtracking search procedure using such static symmetry breaking constraints can be exponentially faster than dynamic methods like SBDD which eliminate all symmetric branches. This is because the symmetry breaking constraints can interact with the problem constraints. We also considered how such symmetry breaking methods can be extended to deal with set variables. Finally, we have tested these symmetry breaking constraints experimentally for the first time and shown that they are effective in practice.

References

1. Puget, J.F.: On the satisfiability of symmetrical constrained satisfaction problems. Proc. of ISMIS'93. (1993) 350–361
2. Crawford, J., Luks, G., Ginsberg, M., Roy, A.: Symmetry breaking predicates for search problems. In: Proc. of the 5th Int. Conf. on Knowledge Representation and Reasoning, (KR '96). (1996) 148–159

3. Flener, P., Frisch, A., Hnich, B., Kiziltan, Z., Miguel, I., Pearson, J., Walsh, T.: Breaking row and column symmetry in matrix models. In: 8th Int. Conf. on Principles and Practices of Constraint Programming (CP-2002), (2002)
4. Law, Y., Lee, J.: Global constraints for integer and set value precedence. In: Proc. of 10th Int. Conf. on Principles and Practice of Constraint Programming (CP2004), (2004) 362–376
5. Fahle, T., Schamberger, S., Sellmann, M.: Symmetry breaking. In: Proc. of 7th Int. Conf. on Principles and Practice of Constraint Programming (CP2001), (2001) 93–107
6. Gent, I., Smith, B.: Symmetry breaking in constraint programming. In Horn, W., ed.: Proc. of ECAI-2000, IOS Press (2000) 599–603
7. Roney-Dougal, C., Gent, I., Kelsey, T., Linton, S.: Tractable symmetry breaking using restricted search trees. In: Proc. of ECAI-2004, IOS Press (2004)
8. Walsh, T.: Breaking value symmetry. Technical Report COMIC-2007-008, NICTA and UNSW (2007)
9. Flener, P., Pearson, J., Sellmann, M., Hentenryck, P.V.: Static and dynamic structural symmetry breaking. In: Proc. of 12th Int. Conf. on Principles and Practice of Constraint Programming (CP2006), (2006)
10. Sellmann, M., Hentenryck, P.V.: Structural symmetry breaking. In: Proc. of 19th IJCAI, (2005)
11. Pesant, G.: A regular language membership constraint for finite sequences of variables. In: Proc. of 10th Int. Conf. on Principles and Practice of Constraint Programming (CP2004), (2004) 482–295
12. Quimper, C.G., Walsh, T.: Global grammar constraints. In: 12th Int. Conf. on Principles and Practices of Constraint Programming (CP-2006), -Verlag (2006)
13. Frisch, A., Hnich, B., Kiziltan, Z., Miguel, I., Walsh, T.: Propagation algorithms for lexicographic ordering constraints. Artificial Intelligence **170** (2006) 803–908
14. Walsh, T.: Symmetry breaking using value precedence. In: Proc. of the 17th ECAI, IOS Press (2006)
15. Quimper, C., van Beek, P., Lopez-Ortiz, A., Golynski, A.: Improved algorithms for the global cardinality constraint. In: Proc. of 10th Int. Conf. on Principles and Practice of Constraint Programming (CP2004), (2004)
16. Puget, J.F.: Breaking all value symmetries in surjection problems. In: Proc. of 11th Int. Conf. on Principles and Practice of Constraint Programming (CP2005), (2005)
17. Law, Y.: Using Constraints to Break Value Symmetries in Constraint Satisfaction Problems. PhD thesis, Department of Computer Science and Engineering, The Chinese University of Hong Kong (2005)
18. Shlyakhter, I.: Generating effective symmetry-breaking predicates for search problems. In: Proc. of LICS workshop on Theory and Applications of Satisfiability Testing (SAT 2001). (2001)
19. Puget, J.F.: Breaking symmetries in all different problems. In: Proc. of 19th IJCAI, (2005) 272–277
20. Law, Y., Lee, J.: Symmetry Breaking Constraints for Value Symmetries in Constraint Satisfaction. Constraints **11** (2006) 221–267

Maintaining Arc Consistency within an intelligent backtracking based informed algorithm

Boutheina Jlifi, Khaled Ghédira

LI3-ENSI, Tunisia

Abstract: This paper introduces maintaining-arc-consistency to the min-conflicts based informed-backtracking algorithm, and shows significant performance gains. Informed backtracking is an important algorithm, because it offers an effective procedure for solving hard constrained combinatorial problems. It combines backtracking with local search, by systematically searching in the space of full assignments, so it gains some of the benefits of local search (i.e. it is "informed" by how close the current assignment is to a solution), but is complete. The problem with all backtracking algorithms is that they can get caught searching in areas with no solution, and that they can take a long time to report that no solution exists. The addition of MAC addresses this, by reducing the time to return a solution on hard problems, and so this paper is a valuable contribution. Two main enhancements of this basic scheme have been proposed: The first new algorithm may be viewed as a strengthening of the IBt process by maintaining full arc consistency. The resulting algorithm is referred to as informed maintaining arc consistency (IMAC). The second enhancement consists in doing intelligent backtracks when the search leads to a dead-end and to integrate the constraint propagation within this backjumping process. To show the advantages of these approaches, experimental comparisons between these latters and other efficient ones in the literature are given. The main contribution of this paper is to incorporate backjumping and maintaining arc consistency in the Informed Backtracking Algorithm.

Key words: Constraint satisfaction problems, Min-conflict-heuristic, informed backtracking algorithm, Maintaining Arc Consistency algorithm, and backjumping algorithm.

1 Introduction

A constraint satisfaction problem is defined as a triple (X, D, C), which involves three sets:

- a set $X = \{x_1, \ldots, x_n\}$ of n variables,
- a set $D = \{ D_{x1}, \ldots, D_{xn}\}$ of n domains, such that each variable $x_i \in X$ takes its value in its finite domain D_{xi},
- a set $C = \{ C1, \ldots, Cm\}$ of m constraints, such that each constraint $Cj \in C$ involves a subset $Xcj = \{x_{cj1}, \ldots, x_{cjk} \}$ of x and is defined by a subset Rc of

$D_{xc1}\times\ldots\times D_{xck}$ specifying which values of the variables are compatible with each other.

Finding a solution to a CSP, consists in determining an assignment of the variables satisfying all the constraints, or indicating that there is none for this CSP. This task is highly combinatorial and generally NP-Complete. In addition to their simple and generic formalisation, CSPs are omni-present in many real-life problems ranging from school examples such as n-queens and graph colouring problems to industrial applications such as scheduling and planning. One such problem that of scheduling astronomical observations on the Hubble Space Telescope (HST) has been noted (Johnston & Minton, 1994) as the original motivation for the informed backtracking (IBT) algorithm that we try to enhance in this paper.

Most of complete search algorithms over Constraint Satisfaction Problems (CSP) are based on Standard Backtracking. Two main enhancements of this basic scheme have been proposed and discussed in this paper (Johnston & Minton, 1994): first, to take an initial inconsistent assignment for variables in a CSP and incrementally repair constraint violations; second, informed and guided backtrackers using a simple ordering heuristic, i.e., the min-conflict heuristic, until a solution is achieved. The resulting algorithm is referred to as informed backtracking (IBT).

The two enhancements described above are of a great interest in increasing the search efficiency for many reasons. On one hand, a general promising technique for solving combinatorial search problems is to generate an initial sub-optimal solution and then to apply local repair heuristics (Johnston and al.,1990) (Johnston and al., 1994) (Morris, 1991) (Selman et al., 1992) (Sosic & Gu, 1990). Methods based on this technique have met with empirical success on many combinatorial problems, including the traveling salesman and graph partitioning problems (Johnston, 1988). Such methods, referred to as repair-based methods, have proved very successful most notably in problem-solving systems that operate by debugging initial solutions (Selman and al., 1992) and have been successfully extended to constraint satisfaction problems (CSPs) (Ghédira, 1994) (Tsang and al., 1999).

On the other hand, Minton explains (Minton and al., 1992) how the min-conflicts-heuristic can improve hill-climbing and backtracking algorithms by giving respectively hill-climbing repair strategy and backtracking repair strategy (called also informed backtracking). The two search strategies have efficiently replaced the "Guarded Discrete Stochastic" (GDS) network, developed by Johnston and Adorf (Adorf & Johnston, 1990), in SPIKE, a system for scheduling HST. They describe their success in solving some other standard problems; in particular the approaches provided a speedy solution to the million queens problem.

The hill-climbing repair strategy most closely replicates the behaviour of the GDS network but has the disadvantage that it can be stuck at a local maximum and fails to find a solution (Minton and al., 1992). In contrast, the informed backtracking algorithm will either find a solution or report a lack of one. Unfortunately, this is of limited significance for large-scale problems because terminating in a failure can take a very long time (Minton and al., 1992) (West, 1995). In fact, a similar work on weak-Commitment search should be mentioned (Tsang, 1996); it presented another modification of min-conflict heuristic by converting a partial solution into a nogood instead of backtracking.

This paper attempts to describe how the efficiency of such an algorithm can be improved by enriching it with other simple methods without loosing the main aspects of the method as being a combination of a tree search technique and a local repair one. The main objective is that of remaining efficient for the resolution of some of the problems described above.
In the first approach, rather than just eliminating from "future" domains values which are inconsistent with the assignment just done like in the forward checking algorithm (Bessière & Régin, 1996), we propagate the effects in order to maintain the arc consistency by using the Maintaining Arc Consistency algorithm (MAC) during the IBt process. For more details about the informed Forward Checking algorithm see (Jlifi & Ghédira, 2004).

The second contribution consists in doing intelligent backtracks when the search leads to a dead-end and to integrate the constraint propagation within this intelligent process.

Although many works have been done about IBt, nothing as far as we know, has ever been published on maintaining Arc Consistency within IBt or combining intelligent and informed backtracks.

This paper is organized as follows. The next section describes Informed Backtracking foundations. The third section explains how to maintain arc consistency during the IBt process by combining it with the MAC algorithm. The fourth section discusses how to do intelligent backtracking when we fail in a dead-end and how to integrate the Arc Consistency process within this informed backjumping process.

The fifth and final section details both experimental design and results. Finally, concluding remarks, discussion of related works and possible extensions to our hybrid algorithms are proposed.

2 Informed backtracking

```
Program Informed-backtracking (X, D, C)
1.  vars-left := ∅
2.  For each x in X do
3.      v := random (Dx)
4.      vars-left := {(x, v)}∪ vars-left
5.  vars-done :=∅
6.  IBt (vars-left, vars-done, D, C)
```

Fig. 1 – Informed Backtracking algorithm

There are two aspects of the informed backtracking method that distinguish it from the standard backtracking algorithm. First, instead of incrementally constructing a consistent partial assignment (i.e., instantiation), it repairs a complete but inconsistent one by reducing inconsistencies. Thus, it uses information about the current assignment to guide its search, which is not available to a standard backtracking algorithm.

Second, the method is guided by a simple ordering heuristic for repairing constraint violations: the Min-conflicts heuristic. It consists in selecting a variable that is currently participating in a constraint violation, and choosing a new value that minimizes the number of outstanding constraint violations.

The algorithm starts with two sets: vars-left and vars-done. First, vars-left is initialized to a set of random assignments for all the variables (figure 1 lines 2-4), and vars-done is initialized to an empty set (figure 1 line 5).

```
Program IBt (vars-left, vars-done, D, C)
1.  If  (conflicts (vars-left)=true)
2.    then x := take-var  (vars-left)
3.    list  :=  sort-values  (x,  Dx,  vars-left,  vars-
                done,C)
4.    while (list ≠ ∅)
5.       w := pop (list)
6.        vars-done := vars-done ∪{(x, w)}
7.        result := IBt (vars-left \{(x, w)},vars-done,D,
              C)
8.        if  (result ≠ NULL)
9.        then return result
10.       return NULL
11. else return (vars-left ∪ vars-done)
```

Fig. 2 – IBt procedure

```
Program Sort-values (x, Dx, I, R, C)
1.  consistent := true
2.   for each v in  Dx do
3.         if consistent ((x, v), R)
4.           then nbcv := 0
5.           for each (y,w) in I do
6.              if (not-consistent ((x,v),(y, w), C x,y)
7.              then nbcv := nbcv+1
8.    list := list ∪{(v,nbcv}
9.   Queue := sort-ascendant-order-by-nbvc (list)
10. return Queue
```

Fig. 3 – Sort-values procedure

Then, the algorithm begins to perform the IBt process by detecting any conflict that exists (figure 2 line 1). If any instantiation (x, v) is found to have a conflict with any other instantiation in vars-left, it is removed from vars-left (figure 2 line2). Let us mention that the ordering variable heuristic used by *take-var* (figure 2 line2) is a random one that uses the first variable included in a detected conflict.

Then, for all the values v' such that (x, v') is compatible with all the assignments in vars-done, v' is placed in a list and ordered in ascending order according to the number of conflicts that it has with the assignments in vars-left (figure 2 line3).
The sort-values process is detailed in figure 3. Thus, the value with the least number of conflicts will be assigned to x (figure2 line 5), and this instantiation will be pushed into vars-done (figure2 line 6).

If no such value exists, i.e., there is no way to repair a variable in vars-left without violating a previously repaired one in vars-done, backtracking takes place and the alternative values in the previously revised variables will be used.

The process terminates when either no conflict is detected among all assignments in vars-left (figure2 line 11) or all the combinations of instantiations have been tried (figure2 line 10).

Note that informed backtracking ensures completeness by looking at all the combinations of the instantiations whenever necessary.

3 Informed Maintaining Arc Consistency (IMAC)

Constraint propagation has been included in the informed backtracking algorithm by us leading to informed forward checking that increases the search efficiency by allowing branches of the search tree that will lead to failure to be pruned earlier than with simple informed backtracking. We have experimentally shown the advantages of this approach, by comparisons between IBt and IFC, especially in terms of complexity (Jlifi & Ghédira, 2004).

So, why not to perform full arc-consistency that will further reduce the domains and remove possible conflicts?

In this paper, we integrate and maintain the arc consistency by the use of the Maintaining Arc Consistency algorithm during the informed based backtracking process. MAC is nowadays considered as one of the best algorithms for solving CSP (Bessière & Régin, 1996). In the MAC algorithm, rather than just revising each domain corresponding with the value of each instantiated variable, MAC makes the network arc-consistent with respect to the instantiated variables.

Like the IBt process, Informed Maintaining Arc Consistency starts with a complete (but inconsistent) variable assignment because a complete but inconsistent assignment provides more guidance (information) than a partial assignment. We recall that vars-left will denote the set of variables to be assigned and vars-done the set of already assigned variables.

The algorithm (figure 4) performs the main loop that tries to assign values to variables as long as a complete consistent assignment has not been found. It consists in iteratively repairing variable assignments until a consistent solution is found.

The algorithm selects any conflicting variable and assigns it a new value that minimises the number of conflicts with other related variables (figure 4 lines 1-2). When an assignment is done, IMAC has to apply the MAC algorithm (figure 4 line8).

In fact, MAC can be viewed as a strengthening of FC. In figure 5 we detail the AC3-MAC process that is used in the MAC algorithm; Rather than just eliminating

from *"future"* domains values that are inconsistent with an assignment just made, one propagates the effects.

Then, we iteratively check if any arcs are not consistent (figure 5 line 5) by the Revise process, deleting values from domains until arc consistency is attained, as described in figure 5 line 3.

```
Program IMAC (vars-left, vars-done, D, C)
1. If  (conflicts (vars-left) = true)
2.     then x := take-var  (vars-left)
3.     list := sort-values (x, Dx, vars-left, vars-done)
4.     save-domain(D)
5.     repeat
6.       while (list ≠ ∅)
7.        w := pop (list)
8.        exist-null-dom (D) := AC3-MAC (vars-left, (x,w), D, C)
9.           if not (exist-null-domain (D))
10.          then vars-done := vars-done ∪{(x, w)}
11.          empty := 0
12.         result := IMAC (vars-left \{(x, w)},vars-done, D, C)
13.            if  (result ≠ NULL)
14.            then return result
15.          else empty := 1
16.           return NULL
17.           restore-domain (D)
18.   until (empty=0)
19. else return (vars-left ∪ vars-done)
```

Fig. 4 – Informed Maintaining Arc Consistency algorithm

```
 Program AC3-MAC (vars-left, (xc, w), D, C)
1. Q := {(xi, xc) in arcs (C)/ i > c}
2. consistent := true
3. while not (empty (Q)) and consistent
4.  (xk, xm) := select-delete-arc (Q)
5.  if Revise (xk, xm)
6.  then Q:= Q ∪{(xi, xk)/(xi, xk) ∈ arcs (C),i≠k, i≠m and i > c}
7.  consistent := not (empty (D(xk)))
8. return consistent
```

Fig. 5 – AC3 for MAC algorithm

In the remaining, the current domain of a variable corresponds to the set of its values that have no conflict with the other related variables. Let us remark that the domains are restored thanks to the restore-domain function (see figure 4 line 17). In fact, we rely on AC3-MAC technology, as a first form of hybridization with the simplest version of the arc-consistency. We can try further reinforcement of the informed backtracking process with the AC7, AC2001 and other relevant ones (Bessière and al., 2001).

4 MAC within Informed Backjumping algorithm (MAC-IBJ)

We illustrate the weakness of the chronological backtracking that we are establishing when we are in a dead-end by an example. Let us assume that the CSP to be solved has the variables x, y and z, each with domain {1,2,3}.

The constraints to be satisfied are: y <= z, xz >= 4. Let us assume that in the search process the x= 1, y=1 instantiations have taken place. Now when trying to instantiate z, it turns out that for all the possible values for z the xz >= 4 constraint is violated. The chronological backtracking process will reconsider the variable y in vain, ending up in the same dead-end situation, without noticing that the cause of the dead-end is obviously in the value assigned to x, the very variable involved in the violated constraint.

```
Program IBj_MAC (vars-left, vars-done, D, C, level)
 1.If  (conflicts (vars-left)= true)
 2.   then x := take-var  (vars-left)
 3.   tab_level [x] := level
 4.   list := sort-values (x, Dx, vars-left, vars- done,C)
 5.   repeat
 6.   begin
 7.   while ((list ≠ ∅) and level (result) ≠ tab-level [x])
 8.    begin
 9.    w := pop (list)
10.    exist-null-domain (D) := AC3-MAC (vars-left, (x,w), D, C)
11.    if not (exist-null-domain (D))
12.    then vars-done := vars-done ∪{(x, w)}
13.    empty := 0
14.     result := IBj_MAC (vars-left \{(x, w)}, vars-done, D, C, tab-
       level, level)
15.            if  (result ≠ NULL)
16.            then return result
17.            else empty := 1
18.            return NULL
19.           restore-domain (D)
20.    end while
21.    level_bj := seek-level-backjumping (x,D, C, level, tab_level)
22.    level(result) := level_bj
23.    end
24.   until (empty = 0)
25. else return (vars-left ∪ vars-done)
```

Fig. 6 – MAC within Informed Backjumping algorithm (MAC-IBJ)

These shortcomings, due to chronological backtracking and blind constraint checking can be avoided by backtracking to such a variable, which can be the cause of the detected dead-end situation.

In a dead-end situation, some variables in Vars-Left are jumped over, instead of doing chronological backtracking by doing an intelligent backtracking and it can be referred to as to informed backjumping process. Like in the IMAC process, we integrate and maintain the arc consistency by the use of the Maintaining Arc

Consistency algorithm during the informed and intelligent backtracking process (figure 6 line 14). The variable to which the backtrack will be done is assessed by the Seek-level-backjumping process (figure 7) that gives the real cause of the detected dead-end situation.

```
Program Seek-level-backjumping (x, D, C, level, tab_level,
vars-done)
1. for every val of Dx
2. begin
3.    temp := level – 1
4.       no_conflict := 1
5.       for evey  xr in vars-done
6.          begin
7.        if not(consistent (x, val, xr, valxr, C)) begin
8.        level_x :=  tab_level [x]
9.        level_xr := tab_level [xr]
10.       temp := min (temp, level_xr);
11.        no-conflict := 0
12.      end
13.      end
14.      if no-conflict
15.      begin
16.        level  := tab_level [x]-1
17.        else
18.        level  := max (level, temp)
19.      end
20.     end
21.     return level
```

Fig. 7 – Seek-level-backjumping process

In fact, by using this intelligent process (Tsang, 1996) we can avoid blind constraint checking and allow branches of the search tree that will lead to failure to be pruned earlier than with IBt. This reduces the search tree and the overall amount of work done.
As mentioned above, the use of an intelligent backtracking is ensured by Gashing Backjumping. In dead, it is possible to perform another intelligent backtracking and relate it to existing algorithms like Graph Based Backjumping and Conflict Based Backjumping. We can mention similar works that have been done in this purpose such as Maintaining arc consistency with conflict directed backjumping (PROSSER ,1995).

5 Experimentation

The goal of our experimentation is to compare a simple implementation of IBt with another one enriched by Maintaining Arc Consistency algorithm. The first implementation is referred to as Informed Backtracking algorithm (IBt) whereas the

second one as Informed Maintaining Arc Consistency algorithm (IMAC). Following this, experimental comparisons between IBJ-MAC algorithm and the MAC one are performed.

The implementation has been done with Visual C++, an Object Oriented language.

5.1 Experimental design

Our experiments are performed on binary CSP-samples randomly generated. In the context of arc consistency this is an important issue and we can speak about different ways of generation (GENT and al., 2001). In this paper, the generation is guided by classical CSP parameters: number of variables (n), domain size (d), constraint density p (a number between 0 and 100% indicating the ratio between the number of the problem effective constraints and the number of all possible constraints, i.e., a complete constraint graph) and constraint tightness q (a number between 0 and 100% indicating the ratio between the number of forbidden pairs of values (not allowed) by the constraint to the size of the domain cross product). As numerical values, we use n = 45 and d = 45. Having chosen the following values 0.1, 0.3, 0.5, 0.7, 0.9 for the parameters p and q, we obtain 25 density-tightness combinations. For each combination, we randomly generate 10 examples. Therefore, we have 250 examples. Moreover and considering the random aspect of generating the initial instantiation, we have performed 10 experimentations per example and taken the average. For each combination density-tightness, we also take the average of the generated examples. The performance is assessed by the following measure:

Run time: the CPU time requested for solving a problem instance,

Moreover, we use another parameter to show the amount of work performed, called constraint-checks (referred to as Cchecks for brevity). The Cchecks gives the number of times the constraints are checked. The Run time shows the complexity. In order to have a quick and clear comparison of the relative performance of the two approaches, we compute ratios of IBt and IMAC performance using the Run time, and constraint checks as follows:

CPU-ratio = IBt-Run-time/ IMAC -Run-time

Cchecks-ratio = IBt- Cchecks/ IMAC -Cchecks

Thus, IMAC performance is the numerator when measuring the CPU time ratios. Then, any number greater than 1 indicates superior performance by IMAC. Let us mention that the CPU-ratio and Cchecks-ratio concerning IBJ-MAC and MAC are similarly calculated and analyzed.

5.2 Experimental results

Figure 8 and figure 9 show the performance ratios from which we deduce the results as follows:

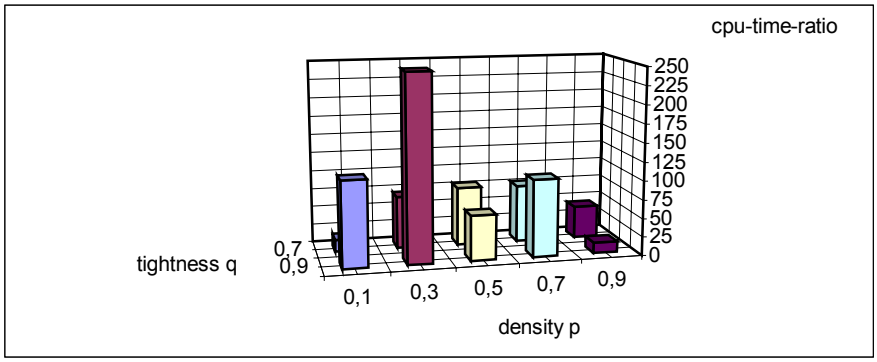


Fig. 8 – CPU ratio

– From the CPU time point of view, IMAC outperforms IBt for all the structured problems we have randomly generated. In fact, IMAC requires up to 225 times less for the tightest set of examples (see figure 8). Furthermore, in the most weakly tight set of examples the performance of IMAC when compared with IBT is not so obvious; the CPU time ratio is about 1 and 1.5 times. This weak deterioration of performance can be explained by the fact that the number of solutions is numerous and the two algorithms do not need so much effort (in term of constraint propagation) for the resolution.

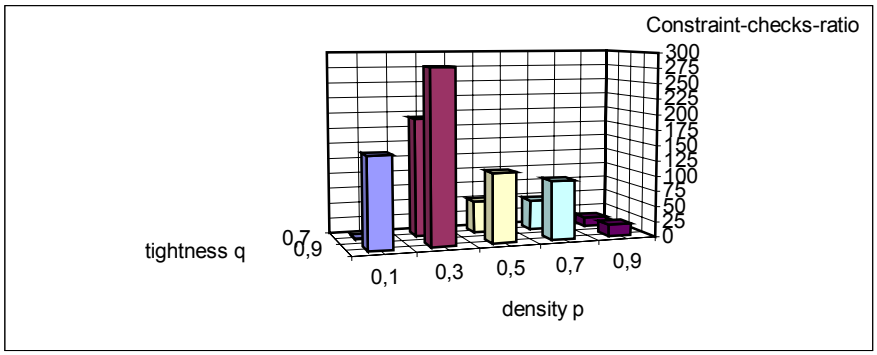


Fig. 9 – Constraint checks ratio

– From the Cchecks-ratio point of view, the IMAC checks much less constraints than IBt. For the tightest set of CSPs, the constraint checks performed by the IBT algorithm are up to 250 times more than IMAC algorithm (figure 9).

Let us mention that the Cchecks-ratio behavior and the CPU time ratio are quite similar for the weakliest tight set of examples and for the tightest ones.

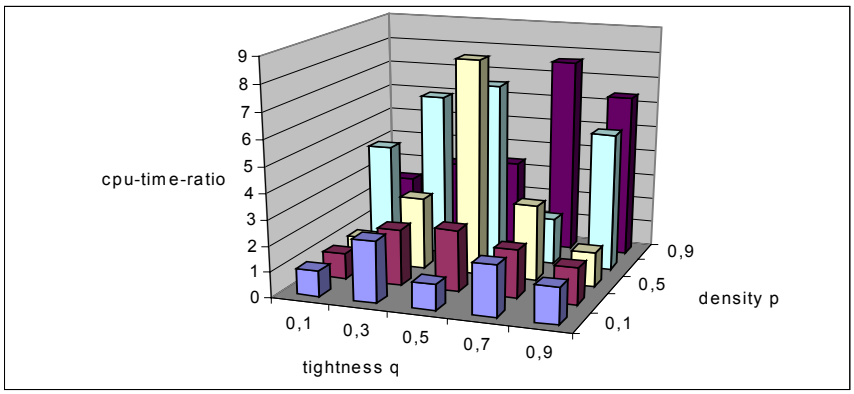


Fig. 10 – CPU ratio

– From the CPU time point of view (figure 10), IBJ-MAC requires up to six times less for the most strongly constrained and, even for the tightest set of examples. Moreover it requires up to three times less for the over-constrained and strongly tight set of examples. Nevertheless, and in some problems, the CPU time required by MAC is always greater or equal to the run time required by IBJ-MAC (respectively in the order of 1.23 or 2.5).

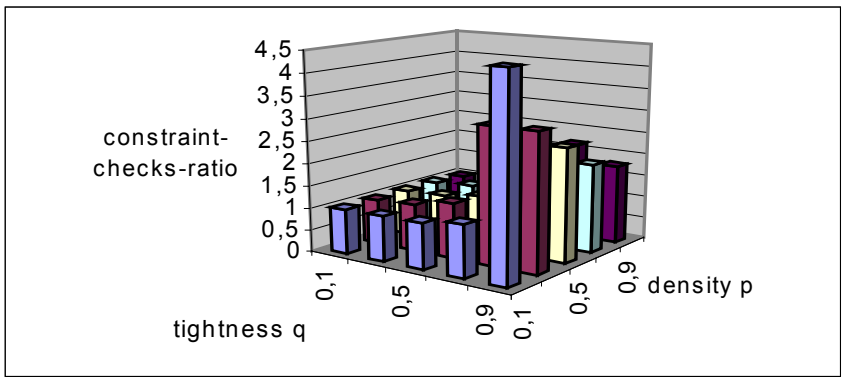


Fig. 11 – Constraint Checks Ratio

– From the Cchecks-ratio point of view, the IBJ-MAC checks much less constraints than MAC, especially for examples where it reveals that there is no solution in a speedy way by pruning search trees that lead to failure earlier than MAC. The obtained results show that the number of constraint checks performed by IBJ-MAC is always less than MAC (figure 11). Thus, IBJ-MAC is distinctly better than MAC

especially in the strongly tight areas (about six times and much more than two in the case of over-constrained problems).

6 Conclusion and Future work

We have enriched the informed backtracking algorithm, which is guided by a simple ordering heuristic for repairing constraint violations: the Min-conflict-heuristic by other efficient methods. Rather than just eliminating from "future" domains values that are inconsistent with an assignment just made, one propagates the effects in order to perform full arc-consistency that will further reduce the domains and remove possible conflicts. Thus, the maintaining of arc consistency within informed backtracking algorithm leads to the Informed Maintaining Arc Consistency algorithm (IMAC). The second enhancement consists in doing intelligent backtracks when the search leads to a dead-end and to integrate the constraint propagation within this backjumping process.

The Experimental comparisons between IBt and IMAC have shown that IMAC outperforms IBt in terms of complexity and quality. In fact, current experimentations are performed and are demonstrating that the new approaches are able to solve very large problems and that they remain stable as the size of the problems arises.

We emphasize that the purpose of this article is to introduce a new way of hybridization rather than to introduce a faster algorithm. We note that this paper attempts to describe how the efficiency of such an algorithm can be improved by enriching it with other simple methods without loosing the main aspects of the method as being a combination of a tree search technique and a local repair one. The main objective is that of remaining efficient for the resolution of some of the problems described and referred in this paper.

Let us mention that experimental comparisons between IBJ-MAC and MAC have given good results in some classes of tightness and density. So, this paper is a valuable contribution since it incorporates backjumping and maintaining arc consistency in the Informed Backtracking Algorithm. As a discussion of related work, like the weak Commitment search mentioned in the introduction to this paper, we should speak about Prestwich'work, which also tries to combine backtracking with local search, and which includes propagation (but his methods are different from ours) (Prestwich 2001).

In order to improve the efficiency of our hybrid algorithms, we can do static or dynamic variable ordering (Bacchus & Van Run,1995) (Tsang, 1996), i.e., use different orderings in different parts of the search tree (also called search rearrangement). We intend to adapt the value ordering heuristic with the filtering method used by using different heuristics in the sort-values procedure. Another way to improve this approach is to change the initial instantiation by another one given by a greedy algorithm. We can perform experiments on real world structured problems (industrial or academic ones). Moreover, we can extend the approach to non-binary CSPs (Bessière and al. 2002).

References

ADORF H.M. & JOHNSTON M.D. (1990). A Discrete Stochastic Neural Network Algorithm for Constraint Satisfaction Problems. In proceedings of the international Joint Conference on Neural Networks, San Diego CA.

BACCHUS F. & VAN RUN P. (1995). Dynamic Variable Ordering in CSPs. In Proceedings the First International Conference on Principle and Practice of Constraint Programming, p. 258-275.

BESSIÈRE C., MESEGUER P. & FREUDER E. (2002). on forward checking for non-binary constraint satisfaction, Larrosa J. Artificial Intelligence 141, p. 205-224.

BESSIÈRE C. & RÉGIN J.C. (1996). MAC and combined heuristics: Two reasons to forsake FC (and CBJ?) on hard problem. In CP'96, Cambridge, MA.

BESSIÈRE C.,RÉGIN J.C., YAP R. H. C & ZHANG Y. (2001). An Optimal Coarse-grained Arc Consistency Algorithm, Artificial Intelligence, volume 165(2), p. 165-185.

GENT I.P., MACINTYRE E., PROSSER P., SMITH B. & WALSH T.(2001). Random Constraint Satisfaction: Flaws and Structure. Journal of Constraints, volume 6(4), p. 345-372.

GHÉDIRA K. (1994). Distributed Simulated Reannealing for Dynamic Constraint Satisfaction Problems. Proceedings of the 6th IEEE International Conference on Tools with Artificial Intelligence, New Orleans USA.

JLIFI B. & GHÉDIRA K. (August 2004). A Study of backtracking based informed algorithms, ECAI04, STAIRS 04, Valence,.

JOHNSTON D.S., PAPADIMITROU C. H. & YANNAKAKIS M. (1988). How easy is local search? Journal of Computer and System Sciences, Vol. 37 p.79-100.

JOHNSTON D.S., ARAGON C.R., MCGEOCH L.A. & SCHEVON C. (1990). Optimization by simulated annealing: An experimental evaluation, part II, Journal of Operations Research.

JOHNSTON M.D. & MINTON S. (1994). Analysing a heuristic strategy for constraint-satisfaction and scheduling. In: Fox, M.S., Zweben, M. (ed.). Intelligent scheduling p. 257-290.

MINTON S., JOHNSTON M., PHILIPS A. B. & LAIRD P. (1990). Solving large scale constraint satisfaction and scheduling problems using a heuristic repair method, In proceedings AAAI-90, Vol. 1 , USA p. 17-24.

MINTON S., JOHNSTON M.D., PHILIPS A.B.& Laird, P. (1992). Minimizing conflicts: A heuristic repair method for constraint-satisfaction and scheduling problems. Artificial Intelligence, Vol. 58 p.161-205.

MORRIS P. (1991). An iterative improvement algorithm with guaranteed convergence, Technical report TR-M91-1, Intellicorp Technical Note.

PRESTWICH S. (2001). Local Search and backtracking vs non systematic Backtracking. Papers from the AAAI 2001 Fall Symposium on Using Uncertainty Within Computation. The AAAI Press. P. 109-115.

PROSSER P. (1995). MAC-CBJ: maintaining arc consistency with conflict directed backjumping. Tech report 95/177.

SADEH N.(1991). Look-ahead Techniques for Micro-Opportunistic Job shop Scheduling. PhD thesis, School of Computer Science, Carnegie-Mellon Univ., Pittsburgh, PA.

SELMAN B., LEVESQUE H. & MITCHELL D. (1992). A new method for solving hard satisfiability problems, in proceedings AAAI-92, San Jose, CA.

SMITH B. M. (1993). Using a Local Improvement Algorithm to solve a constraint satisfaction problem in Roistering. In Juergen Dorn and Karl A. Froeschl, (eds.): scheduling of Production Processes, pages, Chichester, Ellis Horwood p. 94-102.

SOSIC R. & GU J. (1990). A polynomial time algorithm for the n-queens problem, SIGART.

TSANG E.P.K. (1996). Foundations of Constraint Satisfaction Problems, Department of Computer Science, University Of Essex, Colchester, Essex, UK.

TSANG E.P.K, WANG C.J., DAVENPORT A., VOUDOURIS C., LAU T.L (November 1999). A family of stochastic methods for Constraint Satisfaction and Optimization, University of Essex, Colchester, UK.
WEST M. M. (June 1995). Experiments in the evaluation of the Min-Conflicts Heuristic. Technical Report 95.19, School of Computer Studies, University of Leeds.

A generic bounded backtracking framework for solving CSPs

Samba Ndojh Ndiaye and Cyril Terrioux

LSIS - UMR CNRS 6168
Université Paul Cézanne (Aix-Marseille 3)
Avenue Escadrille Normandie-Niemen
13397 Marseille Cedex 20 (France)
{samba-ndojh.ndiaye, cyril.terrioux}@univ-cezanne.fr

Abstract. This paper introduces a new generic backtracking framework for solving CSPs. This scheme exploits semantic and topological properties of the constraint network to produce goods and nogoods. It is based on a set of separators of the constraint graph and several procedures adjustable to exploit heuristics, filtering, backjumping techniques, classical nogood recording, topological (no)good recording, and topological complexity bounds inherited from methods based on graph decompositions like tree-decompositions. According to these choices, we obtain a family of algorithms whose time complexity is between $O(exp(w+1))$ and $O(exp(n))$ with w the tree-width of the constraint graph and n the number of variables.

1 Introduction

The CSP formalism (Constraint Satisfaction Problem) offers a powerful framework for representing and solving efficiently many problems. A CSP consists of a set of variables, which must be assigned in their respective finite domain, by satisfying a set of constraints. Determining if a solution exists is a NP-complete problem.

The usual method for solving CSPs is based on backtracking search, which, in order to be efficient, must use both filtering and heuristic techniques. This approach, often efficient in practice, has an exponential theoretical time complexity in $O(exp(n))$ for an instance having n variables. From a practical viewpoint, FC [1] and MAC [2] are among the most efficient ones. On the other hand, structural methods (e.g. [3–6]) exploit some topological properties of the constraint graph and can thus provide better theoretical time complexity bounds. The best known complexity bounds are given by the "tree-width" of a CSP (often denoted w) and lead to a time complexity in $O(exp(w+1))$ ($w < n$). Unfortunately, the space complexity, often linear for backtracking methods, may make such an approach unusable in practice.

This paper introduces a new generic backtracking framework for solving CSPs. This scheme based on a set of separators of the constraint graph, exploits semantic and topological properties of the constraint network to produce

(no)goods. It uses several adjustable procedures to exploit heuristics, filtering, backjumping techniques and good topological complexity bounds.

This paper is organized as follows. First, we provide the basic notions about CSPs and graphs. Then, we present our generic backtracking framework. Section 4 is devoted to a complexity analysis. Finally, we discuss about related works in section 5 before concluding and outlining future works in section 6.

2 Preliminaries

A *constraint satisfaction problem* (CSP) is defined by a tuple (X, D, C, R). X is a set $\{x_1, \ldots, x_n\}$ of n variables. Each variable x_i takes its values in the finite domain d_{x_i} from D. The variables are subject to the constraints from C. Each constraint c is defined as a set $\{x_{c_1}, \ldots, x_{c_k}\}$ of variables. A relation r_c (from R) is associated with each constraint c such that r_c represents the set of allowed tuples over $d_{x_{c_1}} \times \cdots \times d_{x_{c_k}}$. Given $Y \subseteq X$ such that $Y = \{x_{i_1}, \ldots, x_{i_k}\}$, an *assignment* on the variables of Y is a tuple $\mathcal{A} = (v_{i_1}, \ldots, v_{i_k})$ from $d_{x_{i_1}} \times \cdots \times d_{x_{i_k}}$. We denote by $X_{\mathcal{A}}$ the set of variables assigned in $\mathcal{A}$. An assignment $\mathcal{A}$ is said *partial* if $X_{\mathcal{A}}$ is a subset of X. Given $Y \subseteq X$ and an assignment $\mathcal{A}$, $\mathcal{A}[Y]$ represents the assignment $\mathcal{A}$ restricted to the variables of Y. A constraint c is said *satisfied* by $\mathcal{A}$ if $c \subseteq Y, \mathcal{A}[c] \in r_c$, *violated* otherwise. An assignment is said *consistent* if it does not violate any constraint, *inconsistent* otherwise. Given an instance (X, D, C, R), the CSP problem consists in determining if there is an assignment of each variable which satisfies each constraint. This problem is NP-complete. In this paper, without loss of generality, we only consider binary constraints (i.e. constraints which involve two variables). So, the structure of a CSP can be represented by the graph (X, C), called the *constraint graph*. The vertices of this graph are the variables of X and an edge joins two vertices if the corresponding variables share a constraint. The usual method for solving CSPs is based on backtracking search. The basic backtracking method is chronological Backtracking (denoted BT). It can be significantly improved by using filtering, heuristics, learning or backjumping techniques [7].

Now, we provide some notions about the graph theory. A graph (X, C) is *connected* if there exists a path linking every pair of vertices. Given a subset X' of X, the *subgraph induced by X' from* a graph (X, C) is the graph (X', C') with $C' = \{\{x, y\} \in C,\ x, y \in X'\}$. A connected component of a graph (X, C) is a maximal subset V of X such that the graph induced by V from (X, C) is connected (i.e. there is no subset V' of X such that $V \subset V'$ and the graph induced by V' from (X, C) is connected). Of course, a connected graph has a single connected component. A separator of a connected graph (X, C) is a subset S of X such that the subgraph induced by $X \backslash S$ from (X, C) has at least two connected components. A separator S of a graph (X, C) is said *minimal* if there is no separator S' of (X, C) such that $S' \subset S$. In the connected graph of figure 1(a), the set $\{x_3\}$ is a minimal separator that disconnects the graph into two connected components $\{x_1, x_2, x_4, x_{10}, \ldots, x_{14}\}$ and $\{x_5, \ldots, x_9\}$.

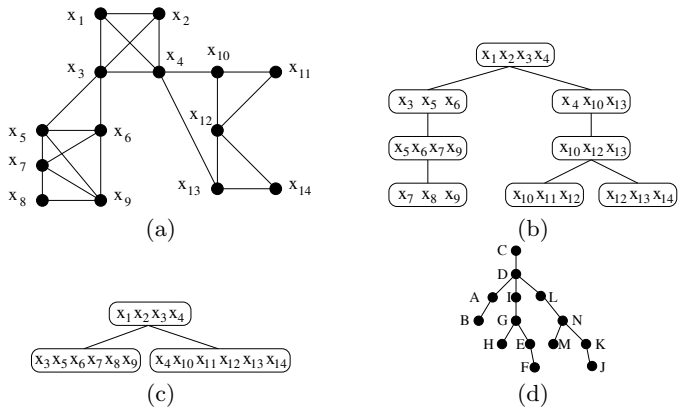


Fig. 1. (a) A graph, (b) a tree-decomposition, (c) a BCC tree, (d) a rooted-tree arrangement / pseudo-tree and (e) a hinge decomposition.

3 A generic backtracking framework

3.1 Theoretical foundations

In this section, we propose a new generic scheme of enumerative algorithms called SBBT (for Separator Based BackTracking). It exploits the separators of the constraint graph of the CSP to record structural (no)goods. Therefore, some parts of the problem will not be visited again since their (in)consistency is known. In this section, we consider a CSP $\mathcal{P} = (X, D, C, R)$ and its constraint graph $G = (X, C)$. Let S_i be a separator of G, CC_{k,S_i} denotes one of the connected components of the subgraph induced by $X \backslash S_i$ from G. A connected overcomponent related to S_i is the set $SP_{k,S_i} = CC_{k,S_i} \cup S_i$. The CC_{k,S_i} sets induce independent subproblems. There is no constraint linking two variables in two independent subproblems. For the graph of figure 1(a), $S_1 = \{x_3\}$ is a separator that disconnects G into two connected components $CC_{1,S_1} = \{x_1, x_2, x_4, x_{10}, \ldots, x_{14}\}$ and $CC_{2,S_1} = \{x_5, \ldots, x_9\}$. The connected overcomponents related to S_1 are $SP_{1,S_1} = \{x_1, x_2, x_4, x_{10}, \ldots, x_{14}, x_3\}$ and $SP_{2,S_1} = \{x_5, \ldots, x_9, x_3\}$.

We can define a directed set of separators by only providing the direction of one separator (root separator): let S_j be a separator directed from SP_{k,S_j}, each other separator S_i of the set is directed from the connected overcomponent SP_{l,S_i} containing S_j. Let S_i be a separator directed from SP_{l,S_i} in a directed set of separators, a directed connected overcomponent related to S_i is a connected overcomponent SP_{t,S_i} related to S_i different from SP_{l,S_i}.

Theorem 1 states that the interactions between the subproblems induced by the connected overcomponents pass through the separator. Thus, assignments on these subproblems are compatible if they are equal on the separator.

Theorem 1 *Let S_i be a separator, SP_{k_1,S_i} and SP_{k_2,S_i} two connected overcomponents related to S_i, $\mathcal{A}_1$ and $\mathcal{A}_2$ two assignments on SP_{k_1,S_i} and SP_{k_2,S_i}, $\mathcal{A}_1$ and $\mathcal{A}_2$ are compatible iff $\mathcal{A}_1[S_i] = \mathcal{A}_2[S_i]$.*

Proof: Since CC_{k_1,S_i} and CC_{k_2,S_i} induce independent subproblems, the compatibility of the two assignments pass through the variables of S_i. Therefore, they are compatible iff they are equal on S_i. □
Let us consider an assignment $\mathcal{A}$ on a separator S_i and a SP_{k,S_i}. Two cases can arise. If $\mathcal{A}$ has no consistent extension on CC_{k,S_i}, the reasons of this inconsistency is only due to constraints joining two variables in CC_{k,S_i} or a variable in S_i and another in CC_{k,S_i}, because CC_{k,S_i} is only connected to the rest of the problem by S_i. So, this assignment on S_i can be considered as a structural nogood since any partial assignment $\mathcal{B}$ s.t. $\mathcal{B}[S_i] = \mathcal{A}$ cannot be extended consistently on CC_{k,S_i}. Likewise, if $\mathcal{A}$ has a consistent extension on CC_{k,S_i}, this assignment on S_i can be considered as a structural good since any partial assignment $\mathcal{B}$ s.t. $\mathcal{B}[S_i] = \mathcal{A}$ can be extended consistently on CC_{k,S_i}.

We define formally below the notions of structural goods and nogoods related to connected overcomponents.

Definition 1 *Let S_i be a separator, a structural good (resp. nogood) related to a connected overcomponent SP_{k,S_i} is a consistent assignment on S_i that can (resp. cannot) be consistently extended on the subproblem induced by CC_{k,S_i}.*

A variable x is said *assignable by a good* $\mathcal{A}$ related to an overcomponent SP_{k,S_i} if $x \in CC_{k,S_i}$. Theorem 2 proves that some parts of the search space can be pruned by structural (no)goods.

Theorem 2 *Let S_i be a separator, $\mathcal{A}$ an assignment on S_i and $\mathcal{B}$ a partial consistent assignment on $X - CC_{k,S_i}$, If $\mathcal{A}$ is a good (respectively a nogood) and $\mathcal{B}[S_i] = \mathcal{A}$, then $\mathcal{B}$ can (resp. cannot) be consistently extended on CC_{k,S_i}.*

Proof: If $\mathcal{A}$ is a good, it can be extended consistently on CC_{k,S_i}. We denote by $Sol_{\mathcal{A},SP_{k,S_i}}$ the solution on SP_{k,S_i} related to the good. Since $\mathcal{B}[S_i] = \mathcal{A}$, $Sol_{\mathcal{A},SP_{k,S_i}}$ and $\mathcal{B}$ are compatible (according to theorem 1). Thus, $\mathcal{B}$ can be extended consistently on CC_{k,S_i}.

If $\mathcal{A}$ is a nogood, it cannot be extended consistently on CC_{k,S_i}. Since $\mathcal{B}[S_i] = \mathcal{A}$, if there is a consistent extension of $\mathcal{B}$ on SP_{kS_i}, it would be a consistent extension of the nogood (according to theorem 1): this is impossible. Thus, there is no consistent extension of $\mathcal{B}$ on CC_{k,S_i}. □

3.2 The generic scheme SBBT

In SBBT, $\mathcal{A}$ denotes the current partial assignment (which is consistent), V the set of unassigned variables, V_g the set of assignable variables thanks to goods, x the current variable, d_x the initial domain of x, d its current domain, v the current value of x, J the set of variables involved in the failures which have occurred during the extension of the current partial assignment. SBBT includes several functions and procedures. $Heuristic_{var}$ is the variable ordering heuristic. It can be defined in different ways to exploit more or less the problem structure. $Heuristic_{val}$ is the value ordering heuristic. *Check_Good_Nogood*$(\mathcal{A}', x, V, V'_g, J)$ checks, for each separator S_j becoming fully assigned in the new current assignment $\mathcal{A}'$, whether $\mathcal{A}'[S_j]$ is a good or a nogood related to a subproblem SP_{k,S_j}. In

Algorithm 1: SBBT(in: $\mathcal{A}, V$, in/out: V_g)

```
1  if V − V_g = ∅ then return ∅
2  else
3      x ← Heuristic_var(V − V_g)
4      d ← d_x; J ← ∅; Backjump ← false
5      while d ≠ ∅ and Backjump = false do
6          v ← Heuristic_val(d)
7          d ← d − {v}; A' ← A ∪ {x ← v}
8          if A' satisfies all constraints then
9              if Check_Good_Nogood (A', x, V, V'_g, J) then
10                 V_g ← V_g ∪ V'_g
11                 Good_Recording(A', x, V, V_g)
12                 J' ←SBBT(A', V − {x}, V_g)
13                 Good_Cancel(x, V, V_g)
14                 if x ∈ J' then  J ← J ∪ J'
15                 else  J ← J'; Backjump ← true
16             else J ← J ∪ Failure(A', x)
17     Nogood_Recording (A, x, V)
18     return J
```

Algorithm 2: Failure(in:$\mathcal{A}', x$)

```
1  return {x} ∪ {y ∉ V | c = {x, y} ∈ C and A' violates c}
```

case $\mathcal{A}'[S_j]$ is a nogood related to SP_{k,S_j}, the variables in SP_{k,S_j} are added to J because SP_{k,S_j} contains the variables causing actually this failure. Then, $false$ is returned meaning that, since $\mathcal{A}'$ contains a nogood, it cannot lead to a solution. In case $\mathcal{A}'[S_j]$ is a good related to SP_{k,S_j}, the variables in SP_{k,S_j} are added to V'_g. This set is returned to SBBT if there is no nogood in $\mathcal{A}'$ and thus they become assignable variables thanks to goods. *Good_Recording*$(\mathcal{A}', x, V, V_g, J)$ records $\mathcal{A}'[S_j]$ as a good related to SP_{k,S_j} for each SP_{k,S_j} becoming fully assigned in the current assignment. *Good_Cancel*(x, V, V_g) removes from V_g all assignable variables thanks to goods containing the variable x whose value is about to be unassigned in SBBT. The procedure *Failure*$(\mathcal{A}', x)$ returns a set of variables containing those that actually cause the failure. *Nogood_Recording*$(\mathcal{A}, x, V, J)$

Algorithm 3: Check_Good_Nogood(in:$\mathcal{A}', x, V$,in/out:V'_g, J)

```
1  V'_g ← ∅
2  forall S_j ∈ Sep s.t. S_j ∩ V = {x} do
3      forall SP_{k,S_j} do
4          switch A'[S_j] do
5              case good related to SP_{k,S_j}
6                  V'_g ← V'_g ∪ CC_{k,S_j}
7              case nogood related to SP_{k,S_j}
8                  J ← J ∪ SP_{k,S_j}; return false
9  return true
```

Algorithm 4: Nogood_Recording (in:$\mathcal{A}, x, V$)

```
1 forall S_j ∈ Sep s.t. S_j ∩ V = ∅ do
2     forall CC_{k,S_j} s.t. x ∈ CC_{k,S_j} do
3         if J ∩ CC_{k,S_j} ≠ ∅ and CC_{k,S_j} ⊆ V then
4             Record A[S_j] as a nogood related to SP_{k,S_j}
```

Algorithm 5: Good_Recording (in:$\mathcal{A}', x, V$, in/out:V_g)

```
1 forall SP_{k,S_j} s.t. SP_{k,S_j} ∩ (V − V_g) = {x} do
2     Record A'[S_j] as a good related to SP_{k,S_j}
3     V_g ← V_g ∪ CC_{k,S_j}
```

records $\mathcal{A}[S_j]$ as a nogood related to SP_{k,S_j} for each separator S_j fully assigned in $\mathcal{A}$ such that $x \in CC_{k,S_j}$ and CC_{k,S_j} is fully unassigned and is involved in the reasons of the failure (in J). For example, the functions and procedures described in algorithms 1-6 propose a possible implementation of our generic scheme SBBT. Of course, they respect the specifications provided above. Note that this implementation defines a new enumerative algorithm.

SBBT solves recursively the subproblem with the inputs $\mathcal{A}$, V and V_g. It relies on a set of separators and the related connected overcomponents. In case this set is directed, only the directed connected overcomponents are considered. It returns $\emptyset$ if the assignment $\mathcal{A}$ admits a consistent extension on V, a set J of variables causing the failures otherwise. $Heuristic_{var}$ chooses the next variable x to assign in V (line 3). If the current domain d of x is not empty, $Heuristic_{val}$ chooses a value v in d. In case the extension $\mathcal{A}'$ of $\mathcal{A}$ is not consistent, *Failure* adds to J the set (or a superset) of variables involved in the failure (line 16) and $Heuristic_{val}$ chooses a new value (if any). If $\mathcal{A}'$ is consistent, *Check_Good_Nogood*$(\mathcal{A}', x, V, V'_g, J)$ returns $false$ if $\mathcal{A}'$ contains a nogood with the current value of x. $Heuristic_{val}$ chooses a new value if the domain is not empty. If no nogood is found, *Check_Good_Nogood* returns $true$ with the set V'_g containing the assignable variables thanks to goods with the current assignment of x. These variables are added in V_g. At line 10, *Good_Recording* records the possible new goods containing x. Then SBBT is recursively called on SBBT$(\mathcal{A}', V - \{x\}, V_g)$. If $\mathcal{A}'$ has no consistent extension, the set J' of variables involved in the failure is returned and the current value of x must be changed. So, first, *Good_Cancel* removes from V_g the assignable variables thanks to goods containing x. If x is involved in the failure, SBBT adds J' to J and a new value is chosen for x (if any). Otherwise, $J = J'$ and a backjump occurs to

Algorithm 6: Good_Cancel (in:x, V, in/out:V_g)

```
1 forall S_j ∈ Sep s.t. S_j ∩ V = {x} do
2     V_g ← V_g − ⋃_k CC_{k,S_j}
```

a variable involved in the failure (according to J). Finally, when the current domain of x is wiped-out or a backjump is triggered, *Nogood_Recording*$(\mathcal{A}, x, V, J)$ records new nogoods containing x (if any) and J is returned.

Theorem 3 *SBBT is sound, complete and terminates.*

Proof: The scheme SBBT is based on BT which is sound, complete and terminates. So, we are going to prove that these properties of BT are not endangered by the pruning thanks to (no)goods and the backjumping of SBBT. A good is recorded when a subproblem induced by a SP_{k,S_i} is fully assigned in the current assignment $\mathcal{A}$. So $\mathcal{A}[S_i]$ has a consistent extension on CC_{k,S_i}. Thus $\mathcal{A}[S_i]$ is a structural good related to the subproblem SP_{k,S_i}. For any assignment $\mathcal{B}$ s.t. $\mathcal{B}[S_i] = \mathcal{A}[S_i]$, we know that $\mathcal{B}$ can be extended consistently on CC_{k,S_i} (theorem 2). So, we can safely continue the search on $V \backslash SP_{k,S_i}$. Regarding the nogood recording, we know that if some variables in CC_{k,S_i} are assigned before all the variables in S_i we cannot record the assignment on S_i as a nogood in case it cannot be extended consistently in CC_{k,S_i}. This is due to the fact that SBBT does not try all the possible assignments on CC_{k,S_i} when it backtracks in S_i. So a nogood is recorded when a separator S_i is fully assigned before any variable in a subproblem induced by a CC_{k,S_i} in the current assignment $\mathcal{A}$ and the reasons we fail in extending $\mathcal{A}$ on CC_{k,S_i} are in the subproblem induced by SP_{k,S_i}. So $\mathcal{A}[S_i]$ cannot be extended consistently on CC_{k,S_i}: $\mathcal{A}[S_i]$ is a structural nogood. For another assignment $\mathcal{B}$ s.t. $\mathcal{B}[S_i] = \mathcal{A}[S_i]$, we know that $\mathcal{B}$ cannot be extended consistently on CC_{k,S_i} (theorem 2). So, we can backtrack because the current assignment cannot lead to a solution. Finally, when SBBT fails to extend consistently the current assignment with the variable x, it backjumps to the last assigned variable in J, the set (or superset) of variables involved in the failure. Since the reasons of this failure are in J, backtracking everywhere else will lead to the same failure. Since the additional prunings does not endanger the properties of BT, SBBT is sound, complete and terminates. □

4 Complexity analysis

The complexity of SBBT depends on the set of separators and the procedures it contains. For instance, BT can be obtained from SBBT by using empty *Good_Recording* and *Nogood_Recording* procedures and a naive *Failure* function returning $X'_{\mathcal{A}}$. A chronological backtrack can lead to encounter several times the same failures. In SBBT, these redundancies can be avoided by defining and backtracking in a set containing the variables causing actually the failures (*Backjump* structure (lines 14-15) and *Failure*). This returned set can be computed in different ways (e.g. formulae of CBJ [8] or GBJ [9]). Furthermore, the set of separators and *Check_Good_Nogood*, *Good_Cancel*, *Good_Recording* and *Nogood_Recording* also reduce the size of the search space by using some structural and semantic properties of the problem. Some parts of the search space will be pruned as soon as their (in)consistency is known. Overall, the variable ordering heuristic (function $Heuristic_{var}$) is extremely important for the efficiency of the

algorithms. Its freedom degree can be bounded more or less to derive benefit from the structure of the problem or the efficiency of dynamic heuristics. It is possible to make several combinations of these techniques in order to define new algorithms and to capture in a very easy way well known ones like BTD [6], BCC [10, 11], pseudo-tree search [12], Tree-solve and Learning Tree-solve [13], AND/OR Search Tree and AND/OR Search Graph [14]. In the following, we will present these methods and the way they can be captured by SBBT.

4.1 Separator set based on a tree-decomposition

BTD (for Backtracking with Tree-Decomposition) relies on a tree-decomposition of the constraint graph. Let $G = (X, C)$ be a graph, a *tree-decomposition* [15] of G is a pair $(E, \mathcal{T})$ where $\mathcal{T} = (I, F)$ is a tree with nodes I and edges F and $E = \{E_i : i \in I\}$ a family of subsets of X, such that each subset (called cluster) E_i is a node of $\mathcal{T}$ and verifies: (i) $\cup_{i \in I} E_i = X$, (ii) for each edge $\{x, y\} \in C$, there exists $i \in I$ with $\{x, y\} \subseteq E_i$, and (iii) for all $i, j, k \in I$, if k is in a path from i to j in $\mathcal{T}$, then $E_i \cap E_j \subseteq E_k$. The width of a tree-decomposition $(E, \mathcal{T})$ is equal to $max_{i \in I}|E_i| - 1$. The *tree-width* w of G is the minimal width over all the tree-decompositions of G. In figure 1(b), we have a possible tree-decomposition of the graph in figure 1(a). BTD assigns the variables w.r.t. an order induced by the considered tree-decomposition of the constraint graph. Moreover, some parts of the search space will not be visited again as soon as their (in)consistency is known. This is possible by using the notion of *structural (no)good*. A good (resp. nogood) is a consistent partial assignment on a set of variables (a separator) that can (resp. cannot) be consistently extended on the part of the CSP located after the separator. The variable order is computed as follows. Let Y be a set of assigned variables, $x_i \in E_i$, if $x_i \in Y$, then $\forall E_j \in E$, such that $i \geq j$ $\forall x_j \in E_j$, $x_j \in Y$. So, BTD assigns a variable $x_i \in E_i$ iff all the variables in clusters preceding E_i in the cluster order are assigned first. Its time complexity is $O(exp(w + 1))$.

SBBT can capture BTD, by using as a directed separator set, the set of cluster intersections in the given tree-decomposition directed from the connected overcomponent containing the root cluster and enforcing the $Heuristic_{var}$ to choose the variable in the same order BTD does. Given a numeration on clusters s.t. E_1 is the root cluster, $Heuristic_{1,var}$ chooses as the next variable to assign $x_i \in E_i$ s.t. all the variables in clusters E_j with $j \leq i$ are already assigned or assignable by a good. So, SBBT records at least the same structural (no)goods BTD does. That allows to guarantee the same time complexity bound.

Theorem 4 *The time complexity of SBBT with the configuration described above is* $O(exp(w + 1))$.

$Heuristic_{2,var}$ is similar to $Heuristic_{1,var}$, but it is allowed to choose the next variable in a whole branch of the tree-decomposition (a branch is path from the root cluster to a leaf). We can consider that the clusters in a same branch

are grouped in a single cluster. And we run $Heuristic_{1,var}$ on this new tree-decomposition whose width is $h-1$, where h is the maximum number of variables in a branch of the basic tree-decomposition.

Theorem 5 *The time complexity of SBBT with the configuration described above is* $O(exp(h))$.

$Heuristic_{3,var}$ is similar to $Heuristic_{1,var}$, but we can choose the next variable among $w + k + 1$ variables in a path included in a branch of the tree-decomposition where k is a constant to parameterize [16].

Theorem 6 *The time complexity of SBBT with the the configuration described above is* $O(exp(2(w + k + 1) - s^-))$, *with* s^- *the minimum size of the cluster intersections.*

4.2 Separator set based on biconnected components

Regarding BCC (for Biconnected Component Backtracking), it relies on the biconnected components of the constraint graph. A *biconnected component* (or *bicomponent*) of a graph G is a maximum subgraph of G which is not disconnected by the removal of any vertex. The graph of bicomponents, obtained by representing each bicomponent as a node, then adding an edge between two components if they share a vertex, is a tree (we suppose that the constraint graph is connected) called the BCC tree of G. In figure 1(c), we have a possible BCC tree of the graph in figure 1(a). BCC is based on this tree whose nodes are *naturally* ordered s.t. the children are greater than their parent. Both DFS and BFS traversals result in a natural ordering. BCC assigns the variables of the problem w.r.t. a static *BCC-compatible* order (compatible with the natural ordering of its BCC tree): the variables in V_i are assigned before those in V_j if V_i and V_j are bicomponents s.t. $i < j$. Given a BCC-compatible ordering, the accessor of a bicomponent is its smallest variable. This variable order allows to avoid some redundancies. In fact, some values of the accessors of the bicomponents are marked if it is known that they can be extended consistently on a subset of the next variables according to the order. So the next time these same values are assigned to those variables, a forward-jump is performed to the unvisited part of the problem. If a value of an accessor cannot be consistently extended on a subset of the next variables according to the order, it is removed from the problem. Moreover, if a failure occurs, BCC backjumps to the last assigned variable whose value could explain this failure. Its time complexity is $O(exp(k))$ with k the size of the largest bicomponent.

SBBT can also capture the BCC method, by using as a directed separator set the set of bicomponent intersections of the given BCC tree, the same set of variables causing the failures in *Failure* and a BCC-compatible variable ordering for $Heuristic_{var}$ ($Heuristic_{BCC,var}$). So, SBBT records at least the values marked (resp. removed) by BCC as structural goods (resp. nogoods). Besides, SBBT performs the same backjumping when a failure occurs as BCC does. That allows to guarantee the same time complexity bound.

Theorem 7 *The time complexity of SBBT with the the configuration described above is $O(exp(k))$ with k the size of the largest bicomponent.*

4.3 Separator set based on a hinge decomposition

The hinge decomposition is based on the notion of hinge set [17]. Let $G = (X, C)$ be a connected graph, $C' \subseteq C$ containing at least two edges, $CC_1, \ldots, CC_m$ the connected components induced by C' of $G' = (X, C - C')$. C' is hinge if for all $i = 1, \ldots, m$, there is an edge $c_i \in C'$ such that $CC_i \cap var(C') \subseteq c_i$ with $var(C')$ the set of variables linked by the edges in C'. A hinge is minimal if it does not contain any other hinge. A hinge decomposition of G is a tree $\mathcal{T}$ that verifies: (i) the nodes of $\mathcal{T}$ are minimal hinges of G, (ii) each edge in C is at least in one node of $\mathcal{T}$, (iii) two neighbouring nodes A and B of $\mathcal{T}$ share exactly one edge $c_i \in C$, $c_i = A \cap B$, (iv) the variables in the intersection between two nodes of the tree $\mathcal{T}$ are in each node in the path linking these two nodes. The Hinge width (denoted w_H) of a constraint graph G is equal to the maximum size of the nodes in a hinge decomposition: it is an invariant of G named cyclicity degree. Indeed, for a given hinge decomposition, the nodes of the tree are minimal hinges. They define connected components CC_i separated from the rest of the problem by an unique edge c_i. These c_i can be considered as separators. In the framework of binary CSPs, a hinge decomposition can be seen as a tree-decomposition by replacing the set of edges in each node of the tree by the set of variables linked by these edges. Thus, the intersections between the nodes of the tree are separators of the constraint graph. So, SBBT can use the structure derived from a hinge decomposition of the constraint graph in the same way it does with a tree-decomposition. The intersections between nodes of the tree form the directed separator set like previously for the BTD method. It is also possible to use the heuristic $Heuristic_{1,var}$ defined for BTD. The complexity of SBBT is given by the following theorem.

Theorem 8 *The time complexity of SBBT with the configuration described above is $O(exp(w_H))$.*

4.4 Separator set based on a pseudo-tree or on a rooted-tree arrangement

The Pseudo-Tree Search method (PTS [12]) uses the notion of pseudo-tree (figure 1(d)) which allows to take in account the structure of the problem: as soon as some parts of the problem become independent during the solving, they are solved independently. A pseudo-tree $\mathcal{T} = (X, C')$ of $G = (X, C)$, is a directed rooted tree such that each edge in C which is not included in C' links a vertex in X with one of its ancestors in $\mathcal{T}$. The variables are assigned w.r.t. an order induced by $\mathcal{T}$: the solving begins at the root and the subproblems rooted on the sons of the current variable are solved recursively and independently.

The Tree-Solve method [13] is very close to PTS and relies on the notion of rooted-tree arrangement [18]. A rooted-tree arrangement (figure 1(d)) of a graph

$G = (X, C)$, is a directed rooted tree $\mathcal{T} = (X, C')$ such that two neighbouring vertices of G are in a same branch of $\mathcal{T}$ (which is a path from the root to a leaf of the tree). Tree-Solve proceeds in the same way PTS does on a rooted-tree arrangement of the constraint graph.

The AND/OR Search Tree method [14] relies on the computation of an AND/OR search space based on a pseudo-tree of the constraint graph. The independences between the produced subproblems allow to reduce exponentially the size of the search space. Let $\mathcal{T} = (X, C')$ be a pseudo-tree of $G = (X, C)$, the AND/OR search tree related to $\mathcal{T}$, $S_{\mathcal{T}}(\mathcal{P})$ has alternating levels of AND and OR nodes. The OR nodes x_i are variables while the AND nodes $< x_i, v_i >$ (or v_i) correspond to the values assigned to variables in their domain. The root of the AND/OR tree is the node OR given by the root of $\mathcal{T}$. An OR node x_i has a child AND node $< x_i, v_i >$ iff $< x_i, v_i >$ is consistent with the partial assignment defined on the path from the root of the tree to the node x_i. An AND node $< x_i, v_i >$ has a child OR node x_j iff x_j is a son of x_i in the pseudo-tree. A solution of $\mathcal{P}$ is a subtree of the AND/OR search tree containing its root and that verifies: if it contains an OR node then it also contains at least one of its children, if it contains an AND node then it contains all its children and all its leaves are consistent. The AND/OR Search Tree solving method is based on the computing of a pseudo-tree of the constraint graph and the construction of the related AND/OR search tree. Thus, a depth first search to find a solution subtree is sufficient to solve the problem.

SBBT captures PTS, Tree-solve and AND/OR Search Tree by using a variable ordering heuristic induced by a pseudo-tree (PTS and AND/OR Search Tree) or a rooted-tree arrangement (Tree-Solve) of the CSP constraint graph. Besides, the procedures *Good_Recording* and *Nogood_Recording* are defined empty and the function *Failure* returns $X_{\mathcal{A}'}$. The set of separators can be chosen freely.

Theorem 9 *The time complexity of SBBT with the configuration described above is $O(exp(h))$ with h the depth of the pseudo-tree or the rooted-tree arrangement.*

The Tree-Solve and AND/OR Search Tree methods can be improved by recording informations which allow to avoid many redundancies and so to reduce the size of the search space. The notion of parent-separators defined in [14] for a pseudo-tree is quasi-similar to one of definition set of a subproblem for a rooted-tree arrangement [13]. These two notions define a separator set of the constraint graph. For a node x_i in $\mathcal{T}$, a pseudo-tree or a rooted-tree arrangement, the parent-separators set of x_i contains x_i and its ancestors in $\mathcal{T}$ which are neighbours in G of its descendants in $\mathcal{T}$ while the definition set of x_i includes only these ancestors. Identical assignments on a separator lead to the solving of the same subproblem. To avoid these redundancies, it is possible to record these assignments ((no)goods: Learning Tree-solve). For an AND/OR search tree, it has been proved in [14] that two nodes with the same parent-separators set root identical subproblems if the assignments on the variables of the parent-separators set are the same. So it is possible to merge these nodes, this operation leads to a fix-point named minimal context AND/OR search graph (AND/OR Search Graph).

While the basic Tree-Solve and AND/OR Search Tree methods have a linear space complexity, these versions have an exponential space complexity in the induced width w^* of the pseudo-tree or rooted-tree arrangement. Let $G = (X, C)$ be a graph and $\mathcal{T} = (X, C')$ a pseudo-tree or a rooted-tree arrangement of G, the induced width of $\mathcal{T}$ is the width of $G = (X, C \cup C')$. For a given order on nodes of the tree, the width of a node is the number of its neighbours preceding it in the order. The width of the order is the maximum width over all nodes. The width of a graph is the minimum width over all possible orders.

SBBT captures the Learning Tree-Solve and the AND/OR Search Graph methods by using as directed separator set, the set of subproblem definition sets induced by the rooted-tree arrangement (Learning Tree-Solve) or the set of parents-separators induced by the pseudo-tree (AND/OR Search Graph) and a variable ordering induced by a prefix numeration on the tree for $Heuristic_{var}$. This time, the procedures *Good_Recording* and *Nogood_Recording* and the function *Failure* must be defined in the usual way. The (no)goods recorded on the separators are the same recorded by the Learning Tree-Solve method. They constitute as well the set of merged nodes in the minimal context graph of the AND/OR Search Graph method.

Theorem 10 *The time complexity of SBBT with the configuration described above is* $O(exp(w^*))$.

4.5 General case

We see that SBBT can easily capture several well known methods. Furthermore, it defines a family of new methods like the possible implementation proposed in section 3. This new scheme allows to compute directly a set of separators and so to ensure some suitable properties on it (e.g. the separator size or number, or the connected component size). Since a set of separators defines a family of tree-decompositions, it gives a more general structure. It is also easier to compute a structure with suitable properties than for tree-decompositions on which an additional work must often be performed to obtain these properties. The SBBT scheme also uses backjumping techniques, the notions of structural (no)goods to reduce the size of the search space by avoiding many redundancies. Besides, the $Heuristic_{var}$ has a significant impact on the number of (no)goods recorded. Unlike methods like BTD or BCC which limit the possible $Heuristic_{var}$, SBBT gives a total freedom in this choice and continues to exploit (no)goods. Yet, we have no guarantee on the number of structural (no)goods recorded by SBBT. So, it is not possible to preserve good theoretical time complexity bounds that depend on the redundancies avoided by using the (no)goods. In practice, it may be possible to record a considerable number of (no)goods, but theoretically, we have the same time complexity as BT.

Theorem 11 *In the general case, the time complexity of SBBT is* $O(exp(n))$.

The space complexity of SBBT only depends on the separator set, since all the informations recorded are assignments on the separators. The number of

(no)goods recorded on a separator S_i is bounded by $d^{|S_i|}$. Thus, the memory space required is bounded by the maximum number of (no)goods that can be recorded on the separators.

Theorem 12 *Let s be the maximum size of the separators, the space complexity of SBBT is $O(n.s.exp(s))$.*

We show that the time complexity bound of SBBT depends on the separator set, the variable ordering heuristic and on the functions and procedures used. Furthermore, according to the choices, we have seen that SBBT is able to capture in different ways several well known methods exploiting the problem structure.

5 Related works

The generic framework we propose in this paper allows us to cover a large spectrum of algorithms according to the choice made for the separator set and the included procedures and functions. This spectrum includes algorithms from structural methods (e.g. BTD, BCC, PTS, Tree-Solve, Learning Tree-Solve, AND/OR Search Tree, AND/OR Search Graph) to backtracking ones like BT. Moreover, whereas the SBBT presentation is based on BT, we can safely exploit filtering techniques which do not modify the constraint graph. For instance, SBBT can rely on FC or MAC. Yet, a filtering like path-consistency cannot be used since it may add some constraints and so some separators may not remain separators of the new constraint graph.

We show as well that SBBT can easily capture GBJ [9] and CBJ [8] by defining the function *Failure* in the right way. Regarding learning algorithms, SBBT turns to be related to the Nogood Recording algorithm (NR [19]). In fact, the structural nogoods of SBBT are a special case of classical nogoods exploited in NR. They mostly differ in their justifications. For structural nogoods, the justifications rely on the separators and the induced subproblems (i.e. on the structure of the constraint graph) instead of the encountered conflicts for classical nogoods.

Finally, the spectrum covered by SBBT includes structural methods. For instance, SBBT captures PTS and AND/OR Search Tree if the variable ordering is induced by a pseudo-tree of the constraint graph, Tree-Solve if it is induced by a rooted-tree arrangement. In case the set of separators is computed from a tree-decomposition of the constraint graph, SBBT is equivalent to BTD. If this set is based on biconnected components of the constraint graph, it is equivalent to BCC. Likewise, our generic framework captures the Learning Tree-solve method if the set of separators is computed from a rooted-tree arrangement and the AND/OR Search Graph method in case the separator set is computed from a pseudo-tree. Nevertheless, while the most of structural methods exploit static variable orders, SBBT does not suffer from this drawback. It results that the time complexity and the ability to record nogoods depend on the degree of freedom of the used variable ordering. Indeed, nogoods are only recorded when this recording is safe, what may decrease the number of recorded nogoods w.r.t.

structural methods which exploits static variable order like BTD or BCC. Note that the recording of goods is independent of the variable order.

6 Conclusion and future works

In this paper, we have described a generic framework called SBBT which exploits semantic and topological properties of the constraint network to produce (no)goods. In particular, SBBT exploits a separator set of the constraint graph. It can be modulated to exploit heuristics, filtering, classical nogood recording, topological (no)good recording, and topological complexity bounds inherited from graph decompositions like tree-decompositions. By so doing, the spectrum of algorithms described by SBBT includes algorithms from structural methods (e.g. BTD, BCC, PTS, Tree-Solve, Learning Tree-Solve, AND/OR Search Tree, AND/OR Search Graph) to backtracking ones like BT. Hence, the time complexity varies between $O(exp(w+1))$ and $O(exp(n))$ for a constraint graph having a tree-width w and n variables. The space complexity is $O(n.s.exp(s))$ with s the size of the largest separator.

Even if the time complexity of SBBT depends on the used separator set and variable ordering heuristic, SBBT does not require any particular feature for the separators. In other words, any set of separators may be exploited in SBBT. Yet, if the separator set relies on some topological properties of the constraint graph (e.g. a tree-decomposition or bicomponents), we can obtain a more powerful algorithm with a better time complexity bound. As no condition is required on the separator set, we may easily derive hybrid algorithms exploiting different topological features according to the considered part of the constraint graph. For instance, on one part of the problem, the separators can be computed from a tree-decomposition and on the other from bicomponents.

Furthermore, the exploited variable ordering heuristic has also an influence on the ability to record nogoods. The more free the heuristic is, the less structural nogoods are recorded. As the recorded nogoods allow SBBT to avoid some redundancies, their recording and use may have a significant impact on the solving efficiency. Likewise, it is well known that variable ordering heuristics play a central role in the efficiency of solving methods. So, from a practical viewpoint, it could be interesting to exploit some trade-off between the freedom of the variable ordering heuristic and the ability of recording structural nogoods. Our generic framework is powerful enough to capture such trade-offs.

Concerning the future works, the influence of the variable heuristic on the ability to record safe nogoods must be reduced. A solution could rely on the exploitation of some techniques from Dynamic Backtracking [20]. Then, we must compare SBBT to other structural or backtracking methods. Regarding the separator set, in this article, SBBT is presented by using a static separator set which is computed before the solving. So a promising extension consists in computing it dynamically. Finally, it could be useful to extend this work to optimization problems modeled as soft constraints [21].

References

1. R. Haralick and G. Elliot. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980.
2. D. Sabin and E. Freuder. Contradicting Conventional Wisdom in Constraint Satisfaction. In *Proc. of ECAI*, pages 125–129, 1994.
3. R. Dechter and J. Pearl. Tree-Clustering for Constraint Networks. *Artificial Intelligence*, 38:353–366, 1989.
4. G. Gottlob, N. Leone, and F. Scarcello. A Comparison of Structural CSP Decomposition Methods. *Artificial Intelligence*, 124:343–282, 2000.
5. G. Gottlob, M. Hutle, and F. Wotawa. Combining hypertree, bicomp and hinge decomposition. In *Proc. of ECAI*, pages 161–165, 2002.
6. P. Jégou and C. Terrioux. Hybrid backtracking bounded by tree-decomposition of constraint networks. *Artificial Intelligence*, 146:43–75, 2003.
7. R. Dechter. *Constraint processing*. Morgan Kaufmann Publishers, 2003.
8. P. Prosser. Hybrid Algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9:268–299, 1993.
9. R. Dechter. Enhancement Schemes for Constraint Processing: Backjumping, Learning, and Cutset Decomposition. *Artificial Intelligence*, 41:273–312, 1990.
10. J.-F. Baget and Y. Tognetti. Backtracking Throught Biconnected Components of a Constraint Graph. In *Proc. of IJCAI*, pages 291–296, 2001.
11. E. Freuder. A Sufficient Condition for Backtrack-Bounded Search. *JACM*, 32:755–761, 1985.
12. E. Freuder and M. Quinn. Taking Advantage of Stable Sets of Variables in Constraint Satisfaction Problems. In *Proc. of IJCAI*, pages 1076–1078, 1985.
13. R. J. Bayardo and D. P. Miranker. A Complexity Analysis of Space-Bounded Learning Algorithms for the Constraints Satisfaction Problem. In *Proc. of AAAI*, pages 298–304, 1996.
14. R. Dechter and R. Mateescu. AND/OR search spaces for graphical models. *Artificial Intelligence*, 171:73–106, 2007.
15. N. Robertson and P.D. Seymour. Graph minors II: Algorithmic aspects of treewidth. *Algorithms*, 7:309–322, 1986.
16. P. Jégou, S.N. Ndiaye, and C. Terrioux. 'Dynamic Heuristics for Backtrack Search on Tree-Decomposition of CSPs. In *Proc. of IJCAI*, pages 112–117, 2007.
17. M. Gyssens, P. Jeavons, and D. Cohen. Decomposing constraint satisfaction problems using database techniques. *Artificial Intelligence*, 66:57–89, 1994.
18. F. Gavril. Some NP-complete Problems on Graphs. In *Proc. of the Conference on Information Sciences and Systems*, pages 91–95, 1977.
19. T. Schiex and G. Verfaillie. Nogood Recording for Static and Dynamic Constraint Satisfaction Problems. *IJAIT*, 3(2):187–207, 1994.
20. M. Ginsberg. Dynamic Backtracking. *Journal of Artificial Intelligence Research*, 1:25–46, 1993.
21. S. Bistarelli, H. Fargier, U. Montanari, F. Rossi, T. Schiex, and G. Verfaillie. Semiring-based CSPs and valued CSPs: Basic properties and comparison. *LNCS*, 1106, 1996.

Reconstructing almost-linear Tree Equation Solving Algorithms in CHR

Marc Meister and Thom Frühwirth

Fakultät für Ingenieurwissenschaften und Informatik
Universität Ulm, Germany
{Marc.Meister,Thom.Fruehwirth}@uni-ulm.de

Abstract. Solving equations over trees is an essential problem in symbolic computation. We reconstruct almost-linear tree equation solving algorithms in the high-level and rule-based Constraint Handling Rules (CHR) language. To this end, we combine the available CHR solver for rational trees with the union-find algorithm. We extend the almost-linear CHR rational tree solver to handle existentially quantified conjunctions of equations in the theory of finite or infinite trees in almost-linear time.

1 Introduction

Static unification, i.e. equation solving over trees, is an essential problem in symbolic computing, in particular in theorem proving and declarative programming languages. Logic programming languages like Prolog rely on unification to treat logical variables, term rewriting systems need it for confluence testing, Constraint Handling Rules (CHR) [5] for both, and functional languages like ML for type checking.

Even though *almost-linear time algorithms* based on the essential union-find algorithm [17] for the unification problem over finite and rational trees are known since the 1970ties, e.g. [9] and [12], they are rarely implemented e.g. in Prolog with the argument that they are too complicated and cause significant overhead.

Moreover, in the context of constraint logic programming, one also needs to deal with local, i.e. existentially quantified variables. It is not obvious how to extend the classic algorithms to these cases without giving up on optimal complexity. In this paper we do so in a straightforward way using CHR as an implementation language. This choice is not accidental. The code in CHR is more concise than even theoretical expositions of unification, the extensions are straightforward. CHR guarantees properties like anytime and online algorithm and is concurrent, and it was shown that any algorithm, including thus union-find, can be implemented in CHR with best-known time and space complexity.

For a lucid exposition of unification algorithms see [1] and for a multidisciplinary survey of unification see [10].

Contributions and Overview. We reconstruct an almost-linear tree equation solving algorithm as concise CHR solver and modify it to solve existentially quantified conjunctions of equations in the theory of finite and infinite trees in almost-linear time.

- We recall the basics of Constraint Handling Rules (CHR) [5], Maher's theory $\mathcal{T}$ of finite or infinite trees [11], and Tarjan's union-find algorithm [17] in Section 2.
- We reconstruct Huet's almost-linear tree solving algorithm for finite and infinite trees [9] in CHR by combining the quadratic classic CHR rational tree solver [5, 13] with the almost-linear CHR union-find solver [15]. Our exceptionally concise, high-level, and rule-based CHR solver has optimal almost-linear time complexity. See Section 3.
- We modify the CHR solver to solve existentially quantified conjunctions of non-flat equations in theory $\mathcal{T}$. The finally solved formula is free of equations that are not linked to the instantiations of free variables. See Section 4.

Supplementary Online Information. Our complete implementation is available online at `http://www.informatik.uni-ulm.de/pm/index.php?id=142`.

2 Preliminaries

Readers familiar with CHR, the theory $\mathcal{T}$, or the union-find algorithm can skip the corresponding sub-section(s).

2.1 Constraint Handling Rules

Constraint Handling Rules (CHR) [5, 16] is a concurrent, committed-choice, rule-based logic programming language. We distinguish between two different kinds of constraints: *built-in constraints* which are solved by a given constraint solver, and *user-defined constraints* which are defined by the rules in a CHR program. This distinction allows one to embed and utilise existing constraint solvers.

A *CHR program* P is a finite set of rules $R \mathbin{@} H_1 \setminus H_2 \Leftrightarrow G \mid B$. Each rule has a unique identifier R, the *head* $H_1 \setminus H_2$ is a non-empty multi-set conjunction of user-defined constraints, the *guard* G is a conjunction of built-in constraints, and the *body* B is a goal. A *goal* is a multi-set conjunction of built-in and user-defined constraints. We omit the trivial guard expression "*true* |". A rule R is a *simpagation rule* if both head expressions H_1 and H_2 are non-empty. If expression H_1 is empty, we have a *simplification rule* and write $R \mathbin{@} H_2 \Leftrightarrow G \mid B$. We do not use *propagation rules* with empty head expression H_2 in this paper.

The *operational semantics* of CHR is defined by a state transition system where states are multi-set conjunctions of atomic constraints. Any one of the rules that are applicable can be applied and rule application cannot be undone since CHR is a committed-choice language. A rule $R \mathbin{@} H_1 \setminus H_2 \Leftrightarrow G \mid B$ is applicable in state $\langle H_1' \wedge H_2' \wedge C\rangle$ if the built-in constraints C_b of C imply that H_1' matches H_1, H_2' matches H_2, and the guard G is entailed under this matching, cf. (1). The consistent, predicate logic, built-in constraint theory CT contains at least Clark's syntactic equality $\doteq$.

$$\begin{array}{ll} \text{IF} & R \mathbin{@} H_1 \setminus H_2 \Leftrightarrow G \mid B \text{ is a fresh variant of rule } R \text{ with new variables } \bar{X} \\ \text{AND} & CT \models (\forall)\, C_b \rightarrow \exists \bar{X}\, (H_1 \doteq H_1' \wedge H_2 \doteq H_2' \wedge G) \\ \text{THEN} & \langle H_1' \wedge H_2' \wedge C\rangle \rightarrowtail_R \langle H_1' \wedge G \wedge B \wedge H_1 \doteq H_1' \wedge H_2 \doteq H_2' \wedge C\rangle \end{array} \tag{1}$$

If applied, a rule *replaces* the matched user-defined constraints of the head expression H_2 in the state by the body of the rule. Rules are applied until exhaustion, i.e. the CHR run-time system computes the reflexive transitive closure $\rightarrowtail_P^*$ of $\rightarrowtail_P$. The derivation $\langle C\rangle \rightarrowtail_P^* \langle C'\rangle$ has initial goal C, answer C', and *derivation length* defined by the number of rule applications. Whenever the conjunction of constraints in a state becomes inconsistent the derivation terminates immediately with answer *false*.

CHR rules have an immediate predicate logic declarative semantics. For a simplification rule, the guard implies a logical equality between the l.h.s. and r.h.s. of the rule. Formally, the logical reading of the simplification rule $R \mathbin{@} H_2 \Leftrightarrow G \mid B$ is $(\forall)\ G \rightarrow (H_2 \leftrightarrow \exists \bar{Y}\ B)$ where $(\forall)$ denotes universal closure and $\bar{Y}$ are the variables that appear only in the body B.

2.2 Theory $\mathcal{T}$ of Finite or Infinite Trees

The theory $\mathcal{T}$ of finite or infinite trees is equivalent to Clark's equality theory (CET) *without* the occur-check (acyclicity) and one *additional uniqueness axiom*, which handles implied equalities, makes the theory complete [11]. The signature of $\mathcal{T}$ consists of an infinite set of distinct function symbols (written as lower-case letters) and the binary predicate symbol $=$.

Besides the usual axioms for *reflexivity*, *symmetry*, and *transitivity* for variables of CET, theory $\mathcal{T}$ has the following axiom scheme according to [11]:

$$(\forall)\ \neg\big(f(S_1,\ldots,S_n) = g(T_1,\ldots,T_m)\big) \tag{A1}$$

$$(\forall)\ f(S_1,\ldots,S_n) = f(T_1,\ldots,T_n) \rightarrow \bigwedge_{i=1}^{n} S_i = T_i \tag{A2}$$

$$(\forall)\ \exists! X_1 \ldots \exists! X_n \bigwedge_{i=1}^{n} X_i = T_i \tag{A3}$$

In (A1), f and g are distinct function symbols. In (A3), $X_1,\ldots,X_n$ are *distinct* variables, $T_1,\ldots,T_n$ are function terms, i.e. no variables, and $\exists! X_i$ denotes that *there exists a unique variable* X_i.

Axiom scheme (A1) is called *contradiction* or *clash* as two distinct function symbols cannot be equal. Axiom scheme (A2) allows to *decompose* an equation by propagating equality to pairwise equality of the arguments. From (A1) and (A2) we see that we can strengthen the implication in (A2) to logical equivalence. The reverse direction is often called *composition*. Axiom scheme (A3) requires that for a particular form of conjunction of equations a *unique* set of solutions exists: For example the formula $\exists X\ X = f(X)$ has a unique solution which is the infinite tree $f(f(f(\ldots)))$. Without (A3), the theory is not complete, e.g. neither does the sentence $\exists X \exists Y\ X = f(X) \wedge Y = f(Y) \wedge \neg(X = Y)$ follow, nor does its negation.

The structure of finite or infinite trees and the structure of the rational trees are models of $\mathcal{T}$. A *rational tree (RT)* is a finite or infinite tree whose set of subtrees is finite, i.e. it has a finite representation as a directed (possibly

cyclic) graph by merging all nodes with common subtrees. A rational tree can be represented as conjunction of binary equality constraints, e.g. the infinite tree $f(f(f(\dots)))$ only contains itself as its set of subtrees $\{f(f(f(...)))\}$ is finite and it can be represented by the equation $X = f(X)$.

The theory $\mathcal{T}$ does not accept full elimination of existential quantifiers, e.g. in the formula $\exists X\ Y = f(X)$ we cannot remove or eliminate the quantifier $\exists X$ and the formula is neither true nor false in $\mathcal{T}$ but depends on the instantiation of the free variable Y.

2.3 Union-Find Algorithm

The classic union-find algorithm solves the problem of maintaining a collection of disjoint sets under the operation of union [17]. Each set is represented by a rooted tree, whose nodes are the elements of the set. The root is called the *representative* of the set. The representative may change when the tree is updated by a union operation. With the algorithm come three operations:

make(X) introduces X by creating a new tree with the only node X;
find(X, R) returns the representative R of the tree in which X is contained by following the path from the node X to the root R of the tree;
union(X, Y) joins the two trees in which X and Y are contained by finding their roots R_X and R_Y. If they are different one root node is updated to point to the other (possibly changing the representative).

With the two independent optimisations *path compression* and *union-by-rank* that keep the trees shallow and balanced, the union-find algorithm has logarithmic worst-case and almost constant amortised running time per operation [17]: For n variables and a mixed sequence of u calls to the union operator and f calls to the find operator, the time complexity for an optimal implementation is $O(m\,G(n))$ with $m = 2u + f$ (we allow calls to union with arguments that are from the same tree). Function G is an extremely slow growing inverse of Ackermann's function with $G(n) < 5$ for all practical n.

Accessing the operations of the union-find algorithm as built-in constraints requires to define ask- and tell-versions for $\text{find}(X, R)$ and $\text{union}(X, Y)$. We define $\text{find}(X, R)$ (ask) to be true iff X is not a root variable. Telling the constraint $\text{find}(X, R)$, however, returns the representative R of the tree in which X is contained. Similarly $\text{union}(X, Y)$ (ask) is true iff X and Y belong to the same tree but only telling $\text{union}(X, Y)$ makes X and Y belong to the same tree.

Clearly the *predicate-logical reading* of $\text{union}(X, Y)$ for two variables is equality $X = Y$. The constraint $\text{union}(X, Y)$ observes the axioms of reflexivity, symmetry, and transitivity of CET for variables: Inserting $\text{union}(X, X)$ keeps the equality sets unchanged and asking $\text{union}(X, X)$ returns true ($X = X \leftrightarrow \mathit{true}$). We have $\text{union}(X, Y)$ iff $\text{union}(Y, X)$, hence the *orientation* of variables is invariant to the built-in theory ($X = Y \leftrightarrow Y = X$). Finally, if $\text{union}(X, Y) \wedge \text{union}(Y, Z)$ holds, then we have $\text{union}(X, Z)$ and union is hence transitive ($X = Y \wedge Y = Z \rightarrow X = Z$). However, $\text{union}(X, Y)$ or $\text{union}(Y, X)$ and the order constraints are told may yield different representatives.

3 Combining the Rational Tree Equation Solver with the Union-Find Algorithm

We reconstruct Huet's almost-linear infinite unification algorithm [9] as a CHR solver accessing Tarjan's union-find algorithm [17] by built-in constraints [2]. We take an *extreme programming* style of development by starting from the classic RT solver [5, 8], add the basic idea to handle equality between variables by the union-find built-in solver, and inspect the necessary changes. We then prove the correctness of our hierarchical solver, show its optimal almost-linear time complexity when using the refined semantics of CHR [3], and briefly explain how to use the optimal CHR union-find implementation [15] as built-in solver.

3.1 Classic CHR Rational Tree Equation Solver

One of the first CHR programs is the classic constraint solver for syntactic equality of rational trees that performs unification [5, 8] where equations $S = T$ between two terms are encoded as CHR constraints S $\mathtt{eq}$ T (cf. Figure 1).

Auxiliary built-ins allow the solver to be independent of the representation of terms. Besides *true* and *false*, we have $\mathrm{v}(T)$ iff T is a variable and $\mathrm{f}(T)$ iff T is a *function term*. Variables are strictly ordered by $\prec$, each variable is smaller than any function term, and function terms are ordered according to term-depth (for details see [13]). The auxiliary $\mathrm{s}(T_1, T_2)$ leads to *false* if T_1 and T_2 have not the same function symbol and the same arity (this is called clash). The auxiliary $\mathrm{a}(T, L)$ returns the arguments of a function term T as a list L.

$$
\begin{aligned}
&\mathit{re} \mathbin{@} X \mathbin{\mathtt{eq}} X \Leftrightarrow \mathrm{v}(X) \mid \mathit{true}\\
&\mathit{or} \mathbin{@} T \mathbin{\mathtt{eq}} X \Leftrightarrow \mathrm{v}(X) \wedge X \prec T \mid X \mathbin{\mathtt{eq}} T\\
&\mathit{de} \mathbin{@} T_1 \mathbin{\mathtt{eq}} T_2 \Leftrightarrow \mathrm{f}(T_1) \wedge \mathrm{f}(T_2) \mid \mathrm{s}(T_1, T_2) \wedge \mathrm{a}(T_1, L_1) \wedge \mathrm{a}(T_2, L_2) \wedge \mathtt{e}(L_1, L_2)\\
&\mathit{co} \mathbin{@} X \mathbin{\mathtt{eq}} T_1 \setminus X \mathbin{\mathtt{eq}} T_2 \Leftrightarrow \mathrm{v}(X) \wedge X \prec T_1 \preceq T_2 \mid T_1 \mathbin{\mathtt{eq}} T_2\\
\mathit{aux} &\mathbin{@} \mathtt{e}([X|L_1], [Y|L_2]) \Leftrightarrow X \mathbin{\mathtt{eq}} Y \wedge \mathtt{e}(L_1, L_2)
\end{aligned}
$$

Fig. 1. Classic rational tree equation solver (RT solver)

We now explain application of each rule of the RT solver:

Reflexivity (*re*) removes trivial equations between identical variables.

Orientation (*or*) reverses the arguments of an equation so that the (smaller) variable comes first.

Decomposition (*de*) applies to equations between two function terms. If there is no clash, the initial equation is replaced by equations between the corresponding arguments of the terms. To this end, the CHR constraint $\mathtt{e}(L_1, L_2)$

pairwise equates the lists of arguments L_1 and L_2 of the two terms using the simple recursion of rule *aux*.[1]

Confrontation (*co*) replaces the variable X in the second equation $X \ \mathtt{eq}\ T_2$ by T_1 from the first equation $X \ \mathtt{eq}\ T_1$. It performs a limited amount of variable elimination (substitution) by only considering the l.h.s.' of equations. The order in the guard ensures termination.

Property 1 ([13]). The classic RT solver terminates and if there is no clash, it returns a conjunction of atomic constraints of the form $\bigwedge_{i=1}^{n} X_i \ \mathtt{eq}\ T_i$ in the theory of the rational trees. The variables $X_1, \ldots, X_n$ are pairwise distinct and X_i is different to T_j for $1 \leq i \leq j \leq n$. For a conjunction of equations with terms of maximal depth one (flat terms) its time complexity is quadratic.[2]

3.2 CHR Program Specialisation to Strict Flat Form

We specialise the classic RT solver w.r.t. goals that are in *strict flat form*.

Definition 1 (Strict Flat). *A conjunction of equations is in* strict flat form *if each equation contains at most one function symbol and each l.h.s. is a variable.*

We apply program transformation techniques for CHR [7]: Two CHR programs P_1 and P_2 are *operationally equivalent*, iff for all states S, we have $S \rightarrowtail^*_{P_1} S_1$, $S \rightarrowtail^*_{P_2} S_2$, and the two final states S_i are identical up to renaming of variables and logical equivalence of built-in constraints.

As the solver decomposes terms, all terms have depth zero or one and partitioning the condition of the guards of *or* and *co* yields the following rules with simplified guards. For conjunctions of equations in strict flat form, the classic RT solver is operationally equivalent to program $\{re, or_1, or_2, de, co_1, co_2, co_3, aux\}$:

$$
\begin{aligned}
&or_1 \ @\ Y \ \mathtt{eq}\ X \Leftrightarrow \mathrm{v}(X) \wedge \mathrm{v}(Y) \wedge X \prec Y \mid X \ \mathtt{eq}\ Y\\
&or_2 \ @\ T \ \mathtt{eq}\ X \Leftrightarrow \mathrm{v}(X) \wedge \mathrm{f}(T) \mid X \ \mathtt{eq}\ T\\
&co_1 \ @\ X \ \mathtt{eq}\ Y \setminus X \ \mathtt{eq}\ Z \Leftrightarrow \mathrm{v}(X) \wedge \mathrm{v}(Y) \wedge \mathrm{v}(Z) \wedge X \prec Y \preceq Z \mid Y \ \mathtt{eq}\ Z\\
&co_2 \ @\ X \ \mathtt{eq}\ Y \setminus X \ \mathtt{eq}\ T \Leftrightarrow \mathrm{v}(X) \wedge \mathrm{v}(Y) \wedge \mathrm{f}(T) \wedge X \prec Y \mid Y \ \mathtt{eq}\ T\\
&co_3 \ @\ X \ \mathtt{eq}\ T_1 \setminus X \ \mathtt{eq}\ T_2 \Leftrightarrow \mathrm{v}(X) \wedge \mathrm{f}(T_1) \wedge \mathrm{f}(T_2) \mid T_1 \ \mathtt{eq}\ T_2
\end{aligned}
$$

To avoid intermediate equations $T_1 \ \mathtt{eq}\ T_2$ with two function terms T_i, we unfold rule *de* into rule co_3 and add the mnemonic rule $de{+}co_3$:

$$
\begin{aligned}
de{+}co_3 \ @\ X \ \mathtt{eq}\ T_1 \setminus X \ \mathtt{eq}\ T_2 \Leftrightarrow \mathrm{v}(X) \wedge \mathrm{f}(T_1) \wedge \mathrm{f}(T_2) \mid\\
\mathrm{s}(T_1, T_2) \wedge \mathrm{a}(T_1, L_1) \wedge \mathrm{a}(T_2, L_2) \wedge \mathtt{e}(L_1, L_2) \ .
\end{aligned}
$$

Rule $de{+}co_3$ short-cuts derivations with intermediate equations $T_1 \ \mathtt{eq}\ T_2$, so we can remove the redundant rules *de* and co_3. As equations are in strict flat form in all states of the derivation, rule or_2 is redundant and we can remove the condition $\mathrm{v}(X)$ from the guard of each rule.

[1] To remove empty constraints $\mathtt{e}([],[])$ one may want to add a rule $\mathtt{e}([],[]) \Leftrightarrow \mathit{true}$.

[2] The classic RT solver also works with non-flat equations with exponential complexity.

$$
\begin{aligned}
re \;@\; & X \;\mathtt{eq}\; X \Leftrightarrow \mathit{true} \\
or_1 \;@\; & Y \;\mathtt{eq}\; X \Leftrightarrow \mathrm{v}(Y) \wedge X \prec Y \mid X \;\mathtt{eq}\; Y \\
co_1 \;@\; & X \;\mathtt{eq}\; Y \setminus X \;\mathtt{eq}\; Z \Leftrightarrow \mathrm{v}(Y) \wedge \mathrm{v}(Z) \wedge X \prec Y \preceq Z \mid Y \;\mathtt{eq}\; Z \\
co_2 \;@\; & X \;\mathtt{eq}\; Y \setminus X \;\mathtt{eq}\; T \Leftrightarrow \mathrm{v}(Y) \wedge \mathrm{f}(T) \wedge X \prec Y \mid Y \;\mathtt{eq}\; T \\
de{+}co_3 \;@\; & X \;\mathtt{eq}\; T_1 \setminus X \;\mathtt{eq}\; T_2 \Leftrightarrow \mathrm{f}(T_1) \wedge \mathrm{f}(T_2) \mid \\
& \qquad\qquad \mathrm{s}(T_1, T_2) \wedge \mathrm{a}(T_1, L_1) \wedge \mathrm{a}(T_2, L_2) \wedge \mathtt{e}(L_1, L_2) \\
aux \;@\; & \mathtt{e}([X|L_1], [Y|L_2]) \Leftrightarrow X \;\mathtt{eq}\; Y \wedge \mathtt{e}(L_1, L_2)
\end{aligned}
$$

Fig. 2. Rational tree solver for strict flat form (RT solver)

Lemma 1 (RT solver for strict flat form). *For conjunctions of equations in* strict flat form*, the classic RT solver (cf. Figure 1) is operationally equivalent to the RT solver for strict flat form (cf. Figure 2).*

Proof. By program specialisation, properties of the order $\prec$, splitting rules according to a partition of the condition of the guards, unfolding, and removing of redundant rules. □

We use the more accessible RT solver for strict flat form for our extreme programming approach.

3.3 An Extreme Programming Development Style

We now want to improve the time complexity of the RT solver for equations in strict flat form. We employ a union-find built-in solver to handle equations between two variables and adapt the RT solver accordingly.

To this end, consider rule *e2u* which replaces equalities $X = Y$ between two variables – encoded by CHR constraints $X \;\mathtt{eq}\; Y$ – with built-in constraints $\mathrm{union}(X, Y)$:

$$e2u \;@\; X \;\mathtt{eq}\; Y \Leftrightarrow \mathrm{v}(Y) \mid \mathrm{union}(X, Y) \;\; .$$

Constraint $\mathrm{union}(X, Y)$ observes the axioms of reflexivity, symmetry, and transitivity of CET for variables (cf. Sub-section 2.3). Rules *re*, or_1, and co_1 implement reflexivity, orientation (a limited form of symmetry), and confrontation between variables (a limited form of transitivity). Taking an extreme programming approach we replace the subsumed rules *re*, or_1, and co_1 with rule *e2u*.

Rule co_2 must be adapted to the union-find data-structure as its head constraint $X \;\mathtt{eq}\; Y$ overlaps with the head of rule *e2u*: We replace the *CHR head constraint* $X \;\mathtt{eq}\; Y$ *of rule* co_2 by the *built-in guard constraint* $\mathrm{union}(X, Y)$:

$$co_2{}' \;@\; X \;\mathtt{eq}\; T \Leftrightarrow \mathrm{union}(X, Y) \wedge \mathrm{f}(T) \wedge X \prec Y \mid Y \;\mathtt{eq}\; T \;\; .$$

In the classic RT solver the strict order of variables $\prec$, guarantees that any function term T is eventually attached to a unique variable Y in the set of equal

variables with $X = Y$. The canonical unique representative in the union-find data-structure for a set of equal variables is its root. Recall that the built-in $\mathrm{find}(X, Y)$ (ask) is true if X is not a root variable, and returns a root variable Y with $X = Y$ when told. Hence, we replace rule co_2 with

$$\mathit{root} \mathbin{@} X \mathbin{\mathtt{eq}} T \Leftrightarrow \mathrm{f}(T) \wedge \mathrm{find}(X, Y) \mid Y \mathbin{\mathtt{eq}} T \ .$$

Note that we dropped $\mathrm{union}(X, Y)$ (ask) from the guard as $\mathrm{find}(X, Y)$ implies $X = Y$. Rules $de{+}co_3$ and *aux* have no $\mathtt{eq}$ constraints between two variables in the head and are not affected by our transformation. We finally unfold rule *e2u* into rule *aux*. We now show that our UF+RT solver, given in Figure 3 is correct.

$$\begin{aligned}
\mathit{e2u} \mathbin{@} X \mathbin{\mathtt{eq}} Y &\Leftrightarrow \mathrm{v}(Y) \mid \mathrm{union}(X, Y) \\
\mathit{root} \mathbin{@} X \mathbin{\mathtt{eq}} T &\Leftrightarrow \mathrm{f}(T) \wedge \mathrm{find}(X, Y) \mid Y \mathbin{\mathtt{eq}} T \\
\mathit{de{+}co_3} \mathbin{@} X \mathbin{\mathtt{eq}} T_1 \setminus X \mathbin{\mathtt{eq}} T_2 &\Leftrightarrow \mathrm{f}(T_1) \wedge \mathrm{f}(T_2) \mid \\
&\qquad \mathrm{s}(T_1, T_2) \wedge \mathrm{a}(T_1, L_1) \wedge \mathrm{a}(T_2, L_2) \wedge \mathtt{e}(L_1, L_2) \\
\mathit{aux}' \mathbin{@} \mathtt{e}([X|L_1], [Y|L_2]) &\Leftrightarrow \mathrm{union}(X, Y) \wedge \mathtt{e}(L_1, L_2)
\end{aligned}$$

Fig. 3. Rational tree solver for strict flat form using union-find (UF+RT solver)

Definition 2 (Solved CHR State). *A CHR state for a built-in theory that includes the union-find is* solved *if it is false or if its CHR constraints are in the form* $\bigwedge_{i=1}^{n} X_i \mathbin{\mathtt{eq}} T_i$ *with pairwise distinct* root variables $X_1, \ldots, X_n$ *and flat functions terms* $T_1, \ldots, T_n$.

Lemma 2 (Correctness). *For conjunctions of equations in strict flat form the UF+RT solver terminates with a solved state in the theory of the rational trees.*

Proof. The solver terminates as rule *e2u* removes CHR constraints $X \mathbin{\mathtt{eq}} Y$ between two variables, rule *root* pushes flat terms equations strictly upwards in the trees, and rule $de{+}co_3$ removes CHR constraints $X \mathbin{\mathtt{eq}} T$ for a function term T. As long as a state is not solved, at least one rule is applicable and if it is in solved form, no rule is applicable.

The logical reading of each rule *e2u*, *root*, and $de{+}co_3$ is valid in theory $\mathcal{T}$ because $X \mathbin{\mathtt{eq}} Y$, $\mathrm{union}(X, Y)$, and $\mathrm{find}(X, Y)$ are encodings for $X = Y$: For rule *e2u* we have $(\forall)\ X = Y \leftrightarrow X = Y$ and for rule *root* we have $(\forall) X = Y \rightarrow (X = T \leftrightarrow Y = T)$. For rule $de{+}co_3$ we consider two cases: If T_1 and T_2 have different function symbols f and g, then $\mathrm{s}(T_1, T_2)$ fails, i.e. $\neg\big(X = f(\ldots) \wedge X = g(\ldots)\big)$, otherwise we have $\big(X = f(X_1, \ldots, X_n) \wedge X = f(Y_1, \ldots, Y_n)\big) \leftrightarrow \bigwedge_{i=1}^{n} X_i = Y_i$ as $\mathtt{e}([X_1, \ldots, X_n], [Y_1, \ldots, Y_n]) \leftrightarrow \bigwedge_{i=1}^{n} X_i = Y_i$ by rule *aux*′. □

Definition 3 (Solved Form). *A conjunction of equations in strict flat form is* solved *if it is false or if it is in the form* $\bigwedge_{i=1}^{n} X_i = T_i$ *with pairwise distinct*

variables $X_1, \ldots, X_n$ *and terms* $T_1, \ldots, T_n$ *for* $n \in \mathbf{N}$*. We require each term* T_i *to be different to* X_j *for* $1 \leq j \leq n$*.*

The formula $X = Y \wedge Z = Y \wedge Y = f(X)$ is solved but $X = Y \wedge Y = Z \wedge Z = f(X)$ is *not solved* as variable Y appears both on the l.h.s. and r.h.s. of equations between variables. We can convert a solved CHR state to a solved form by adding equations $X = R_X$ for each non-root variable X with root-variable R_X in linear time. Hence, root variables are on the r.h.s. in equations between two variables and on the l.h.s. for equations which contain a function symbol.

Lemma 3 (Conversion to Solved Form). *Consider a solved CHR state that is not false with CHR constraints* $\bigwedge_{i=1}^{n} X_i \;\mathtt{eq}\; T_i$ *and conjunction* C_b *of built-ins. Then* $\left(\bigwedge_{X:C_b \to \mathrm{find}(X,R_X)} X = R_X\right) \wedge \left(\bigwedge_{i=1}^{n} X_i = T_i\right)$ *is solved. The amortised time complexity for calling* $\mathrm{find}(X, R_X)$ *for each variable* X *is constant.*

Proof. Calling $\mathrm{find}(X, R_X)$ for each variable X (without intermediate calls to union) touches each node in the trees once due to path compression. □

3.4 Complexity of the UF+RT Solver

As the number of application of rules *e2u*, *de+co*$_3$, and *aux*$'$ is independent of the order rules are applied, we achieve a minimal derivation length when we delay application of rule *root*.

To this end, we use the refined semantics of CHR [3] for scheduling rule and constraint selection. In refined semantics, constraints are inserted sequentially into the store from left-to-right and applicable rules for the constraints in the store are chosen in textual execution order.

$$
\begin{aligned}
\mathit{e2u} \;@\; & X \;\mathtt{eq}\; Y \Leftrightarrow \mathrm{v}(Y) \mid \mathrm{union}(X, Y) \\
\mathit{de{+}co_3} \;@\; & X \;\mathtt{eq}\; T_1 \setminus X \;\mathtt{eq}\; T_2 \Leftrightarrow \mathrm{s}(T_1, T_2) \wedge \mathrm{a}(T_1, L_1) \wedge \mathrm{a}(T_2, L_2) \wedge \mathtt{e}(L_1, L_2) \\
\mathit{aux}' \;@\; & \mathtt{e}([X|L_1], [Y|L_2]) \Leftrightarrow \mathrm{union}(X, Y) \wedge \mathtt{e}(L_1, L_2) \\
\mathit{root} \;@\; & X \;\mathtt{eq}\; T \Leftrightarrow \mathrm{find}(X, Y) \mid Y \;\mathtt{eq}\; T
\end{aligned}
$$

Fig. 4. UF+RT solver for refined semantics (refined UF+RT)

Consider the concise UF+RT solver for refined semantics (cf. Figure 4): Compared to the UF+RT solver, rule *root* is last in textual order and guards are simplified. When rule *root* applies, all equalities of constraints are already propagated, i.e. there are neither equations $X \;\mathtt{eq}\; Y$ between variables nor $\mathtt{e}(L_1, L_2)$ constraints in the store. Also, due to rule *de+co*$_3$ there is at most one equation $X \;\mathtt{eq}\; T$ with a function term T for each variable X in the store. We now bound the number of rule applications.

Lemma 4 (Rule Applications). *Consider a conjunction of equations C in strict flat form with $\#C$ occurrences of variable and function symbols. Then (i) $\#e2u + \#aux' \leq \#C$, (ii) $\#de{+}co_3 \leq \#C$, and (iii) $\#root \leq \#C$ where $\#R$ denotes the number of applications of rule R of the refined UF+RT solver.*

Proof. (i) Consider a measure for conjunctions of CHR constraints $\left|\bigwedge_{i=1}^{n} C_i\right| := \sum_{i+1}^{n} |C_i|$ where $|X \texttt{ eq } T|$ is the number of occurrences of variables in T and $|\texttt{e}(L_1, L_2)|$ is the length of list L_1. Because $|.|$ is invariant to reordering of constraints we can treat local replacements of constraints, caused by a rule applications, independently. The measure is not affected by rules $de{+}co_3$ and *root* and each application of *e2u* or *aux'* decreases the measure by one.[3] Hence $\#e2u + \#aux' \leq |C| \leq \#C$.

(ii) Application of rule $de{+}co_3$ decreases the number of occurrences of function symbols by one and hence we have $\#de{+}co_3 \leq \#C$.

(iii) Consider two cases: When *inserting* $X \texttt{ eq } T$ with a function symbol T to the store, rule *root* applies if X is not a root variable and no other constraint $X \texttt{ eq } T'$ is already in the store. For equations $X \texttt{ eq } T$ that are already *in the store*, rule *root* applies when X is no longer root due to linking. The sum of occurrences of function symbols and the number of variables is bounded by $\#C$. □

We can now give our first main result for an efficient CHR system, e.g. the K.U.Leuven system [14], that allows to find partner constraints for rule application of rule $de{+}co_3$ in *constant time* by using an *index* on the shared variable X of the head $X \texttt{ eq } T_1 \setminus X \texttt{ eq } T_2$. For details on constant time rule selection due to combination of matching, partner constraints, and guards, see [15]. We also require that the built-in union-find algorithm is implemented with optimal almost-linear time complexity.

Theorem 1 (Almost-linear Refined Tree Equation Solver). *Consider an efficient CHR system with indexing and an optimal, almost-linear union-find implementation accessible through built-in constraints. Solving conjunctions of equations in strict flat form with the refined UF+RT solver has almost-linear time complexity.*

Proof. By Lemma 2 the refined UF+RT solver is correct as the refined semantics is an instance of the operational semantics [3]. By Lemma 4, both the number of rule applications and the number of calls to the built-in constraints union (tell) and find (tell) is linear. Also the solver does not introduce new variables. Hence the refined UF+RT solver inherits the almost-linear time complexity of the underlying union-find algorithm. Finally, the solved CHR state is converted in linear time to the solved formula by Lemma 3. □

Theorem 1 improves on the quadratic complexity from [13] to solve equations in the theory of rational trees.

[3] If there is a clash the derivation stops immediately.

3.5 Simulating the Hierarchical UF+RT Solver

The union-find algorithm has been implemented in CHR with optimal, almost-linear time complexity [15]. Because stacking one CHR solver on top of another (cf. [2] for details on hierarchical solvers) is (up to now) not supported by any CHR implementation we are aware of, we cannot use the union-find constraint $\mathrm{find}(X, Y)$ as built-in in the guard of rule *root* directly. To reuse the optimal CHR union-find implementation, where $\mathrm{union}(X, Y)$ and $\mathrm{find}(X, Y)$ are CHR constrains, both constraints can only be accessed in tell-mode. We can simulate the necessary wake-up of rule *root* (when X is no longer a root) of the ask-constraint $\mathrm{find}(X, Y)$ by replacing rule $\mathit{root} \ @ \ X \ \mathtt{eq} \ T \Leftrightarrow \mathrm{find}(X, Y) \mid Y \ \mathtt{eq} \ T$ with

$$\mathit{root}' \ @ \ \mathtt{notroot}(X) \setminus X \ \mathtt{eq} \ T \Leftrightarrow \mathrm{find}(X, Y) \wedge Y \ \mathtt{eq} \ T \ ,$$

where $\mathrm{find}(X, Y)$ is a tell-constraint. We adapt the union-find implementation to insert an CHR constraint $\mathtt{notroot}(X)$ when X becomes a non-root variable due to linking.

4 Existential Variables

In [13] the classic CHR RT solver [5, 8] was modified to solve existentially quantified conjunction of equations with quadratic complexity. We modify the refined UF+RT solver (cf. Figure 4) to solve existentially quantified conjunction of equations in *almost-linear* time.

4.1 Purging Unreachable Variables and Equations

To eliminate existentially quantified variables from an existentially quantified conjunction of equations we require that the conjunction is in *oriented* and *representative* form.

Definition 4 (Oriented Form). *An existentially quantified and solved conjunction of equations* $\exists \bar{X} \ \bigwedge_{i=1}^{n} X_i = T_i$ *is* oriented *if it does not contain equations* $X_j = T_j$ *with a free variable* $X_j \notin \bar{X}$ *and an existentially quantified variable* $T_j \in \bar{X}$.

Any non-oriented, existentially quantified, and solved conjunction of equations can be transformed into an equivalent oriented formula:

Property 2. Consider a solved formula $\exists \bar{X} \ \bigwedge_{i=1}^{n} X_i = T_i$ with an equation $X_j = T_j$ between a free variable X_j and an existentially quantified variable T_j. Then

$$\mathcal{T} \models \left(\exists \bar{X} \ \bigwedge_{i=1}^{n} X_i = T_i\right) \leftrightarrow \left(\exists \bar{X} \ \bigwedge_{i=1}^{n} E_i\right) \text{ with } E_i \equiv \begin{cases} T_j = X_j & \text{if } i = j \\ X_i[X_j \leftarrow T_j] = T_i & \text{if } i \neq j \wedge \mathrm{f}(T_i) \\ X_i = T_i[X_j \leftarrow T_j] & \text{if } i \neq j \wedge \mathrm{v}(T_i) \end{cases}$$

and the conjunction $\bigwedge_{i=1}^{n} E_i$ is in solved form.

Definition 5 (Representative Form). *An existentially quantified, solved, and oriented conjunction of equations* $\exists \bar{X}\ \bigwedge_{i=1}^{n} X_i = T_i$ *is* representative *if each function term* T_j *does not contain an existentially quantified argument variable* $X_k \in \bar{X}$ *with* $k \neq j$.

We can transform an oriented formula to an equivalent representative formula by replacing argument variables by the *representative* variables, i.e. by variables on the r.h.s. of equations between two variables:

Property 3. Consider an oriented formula $\exists \bar{X}\ \bigwedge_{i=1}^{n} X_i = T_i$. Then, we have

$$\mathcal{T} \models \left(\exists \bar{X}\ \bigwedge_{i=1}^{n} X_i = T_i\right) \leftrightarrow \left(\exists \bar{X}\ \bigwedge_{i=1}^{n} X_i = T_i'\right) \text{ with } T_i \equiv \begin{cases} T_i & \text{if } \mathrm{v}(T) \\ T_i\sigma & \text{if } \mathrm{f}(T_i) \end{cases}$$

with $\sigma \equiv \prod_{k:\mathrm{v}(T_k)}[T_k \leftarrow X_k]$ and $\exists \bar{X}\ \bigwedge_{i=1}^{n} X_i = T_i'$ is solved and oriented.

We transform the solved formula $\exists Y\ X = Y \wedge Z = Y \wedge Y = f(Y)$ to the equivalent and oriented formula $Y = X \wedge Z = X \wedge X = f(Y)$. Replacing $X = f(Y)$ by $X = f(X)$ yields the representative formula.

Variables and equations that are linked to to the instantiations of free variables are called *reachable*. Adapting the notion of *reachability* [13] for an existentially quantified conjunction of equations in representative form allows to *purge* non-reachable equations and quantified variables.

Definition 6 (Purged Form). *A formula* $\exists \bar{X}\ \bigwedge_{i=1}^{n} X_i = T_i$ *in representative form is* purged *(or* finally solved*) if all variables in* $\bar{X}$ *and all equations* $X_i = T_i$ *are reachable: A variable* X *is* reachable *if* X *is a free variable or if* X *appears as an argument of a function term* T_i *in a reachable equation* $X_i = T_i$*. An equation* $X_i = T_i$ *is* reachable *if* X_i *is reachable.*

Any non-purged but representative formula can be transformed into an equivalent purged formula by eliminating unreachable equations and variables according to Maher's *uniqueness axiom (A3)*.

Property 4. Consider a formatted formula $\exists \bar{X}\ \bigwedge_{i=1}^{n} X_i = T_i$, its sub-vector $\bar{X}'$ consisting of the reachable variables of $\bar{X}$, and its reachable equations $X_{i_j} = T_{i_j}$. Then we have $\mathcal{T} \models \left(\exists \bar{X}\ \bigwedge_{i=1}^{n} X_i = T_i\right) \leftrightarrow \left(\exists \bar{X}'\ \bigwedge_{j=1}^{k} X_{i_j} = T_{i_j}\right)$.

The representative formula $\exists YUW\ Y = X \wedge Z = X \wedge X = f(W) \wedge W = g(X, W) \wedge U = f(W)$ is equivalent to the purged formula $\exists W\ Z = X \wedge X = f(W) \wedge W = g(X, W)$.

4.2 Transforming to Representative Form and Purging in CHR

To transform a solved form to an equivalent oriented form, we apply program $\{o_1, o_2, o_3\}$ where existentially quantified variables are marked with CHR constraints $\texttt{exists}(X)$, free variables with $\texttt{free}(X)$, and equations $=$ are encoded as CHR constraints $\texttt{eq}$.

$$o_1 \ @\ \texttt{replace}(X,Y) \setminus X\ \texttt{eq}\ T \Leftrightarrow \mathrm{v}(T) \mid Y\ \texttt{eq}\ T$$
$$o_2 \ @\ \texttt{replace}(X,Y) \setminus Z\ \texttt{eq}\ X \Leftrightarrow Z\ \texttt{eq}\ Y$$
$$o_3 \ @\ \texttt{free}(Y) \wedge \texttt{exists}(X) \setminus Y\ \texttt{eq}\ X \Leftrightarrow \texttt{replace}(X,Y) \wedge X\ \texttt{eq}\ Y$$

Under *refined semantics*, rules o_1 and o_2 apply exhaustively for any generated CHR constraint $\texttt{replace}(X, Y)$ by rule o_3 which replaces an equation $Y\ \texttt{eq}\ X$ between a free variable Y and an existentially quantified variable X with $X\ \texttt{eq}\ Y$. Rules o_1 and o_2 update the representatives of all affected equalities for both function terms T attached to Y and for equations $Y\ \texttt{eq}\ X$ between two variables.

To substitute non-representative argument variables and purge non-reachable variables and equations we apply program $\{p_1, p_2, p_3, p_4\}$ on the answer of program $\{o_1, o_2, o_3\}$. The purged (or finally solved) form is encoded by CHR constraints $\texttt{eq}'$ and $\texttt{exists}'$.

$$
\begin{aligned}
&p_1 \ @\ X\ \texttt{eq}\ Y \setminus \texttt{free}(X) \Leftrightarrow \mathrm{v}(Y) \mid X\ \texttt{eq}'\ Y \wedge \texttt{reach}(X)\\
&p_2 \ @\ X\ \texttt{eq}\ T \setminus \texttt{free}(X) \Leftrightarrow \mathrm{f}(T) \mid \texttt{reach}(X)\\
&p_3 \ @\ \texttt{reach}(X) \setminus X\ \texttt{eq}\ T \Leftrightarrow \mathrm{f}(T) \mid \mathrm{reachargs}(T, T') \wedge X\ \texttt{eq}'\ T'\\
&p_4 \ @\ \texttt{reach}(X) \setminus \texttt{exists}(X) \Leftrightarrow \texttt{exists}'(X)
\end{aligned}
$$

Rule p_1 saves equations between two free variables to the finally solved from. Both rules p_1 and p_2 mark free variables X that can lead to other reachable variables with a CHR constraint $\texttt{reach}(X)$. For a reachable variable X, rule p_3 propagates reachability to the arguments of the attached function term T by calling the built-in $\mathrm{reachargs}(T, T')$ which returns a function term T' with representative argument variables and marks the equation as reachable. Rule p_4 saves reachable existentially quantified variables X to the finally solved form.

4.3 Solving Algorithm

Our solving algorithm A for existentially quantified conjunction of non-flat equations $\exists \bar{Y}\ \bigwedge_{i=1}^{n} S_i = T_i$ consists of four sequentially executed parts.

(1) Transform $\exists \bar{Y}\ \bigwedge_{i=1}^{n} S_i = T_i$ to an equivalent existentially quantified conjunction of equations $\exists \bar{X}\ C_1$ in strict flat form by adding new existentially quantified variables (cf. [13] for details).
(2) Apply the refined UF+RT solver on C_1. If the solver terminates with *false* stop with an error, otherwise convert the solved CHR state to the solved form C_2 (cf. Section 3 for details).
(3) Transform $\exists \bar{X}\ C_2$ to an equivalent and oriented formula $\exists \bar{X}\ C_3$ by application of program $\{o_1, o_2, o_3\}$ (cf. Subsection 4.1).
(4) Transform $\exists \bar{X}\ C_3$ to an equivalent finally solved formula $\exists \bar{X}'\ C_4$ by application of program $\{p_1, p_2, p_3, p_3\}$ (cf. Subsection 4.1).

We can now state our second main contribution:

Theorem 2 (Almost-linear Tree Equation Solver With Existential Variables). *The time complexity of algorithm A to solve existentially quantified conjunctions of non-flat equations in theory $\mathcal{T}$ is almost-linear.*

Proof. Flattening can be done in linear time and space and both the number of new existentially quantified variables and the number of new equations are linear in the size of the non-flat existentially quantified conjunction of equations [13]. By Theorem 1 the second step takes almost-linear time for the refined UF+RT solver. Each programs $\{o_1, o_2, o_3\}$ and $\{p_1, p_2, p_3, p_4\}$ traverses the formula once in linear time. □

Our extended CHR solver implements algorithm A with optimal almost-linear time complexity and is available online (cf. link in Section 1).

Example 1. We apply algorithm A on the following formula with the free variable X:

$$\exists VWZ\ W = X \wedge f(X) = f(g(W,Z)) \wedge f(Z) = f(f(V))\ .$$

(1) Flattening to an equivalent formula with additional existential quantified variables and equations in strict flat form yields

$$\exists VWZABCD\ W = X \wedge A = f(X) \wedge B = g(W,Z) \wedge A = f(B) \wedge C = f(Z) \wedge D = f(V) \wedge C = f(D)\ .$$

(2) Application of the refined UF-RT solver on the conjunction of equations which returns a solved form

$$X = W \wedge W = g(W,V) \wedge Z = f(V) \wedge A = f(X) \wedge B = W \wedge C = f(Z) \wedge D = Z\ .$$

(3) Orientation on the quantified and solved formula yields

$$\exists VWZABCD\ W = X \wedge X = g(W,Z) \wedge Z = f(V) \wedge A = f(X) \wedge B = X \wedge C = f(Z) \wedge D = Z\ .$$

(4) Transforming in representative form and purging of unreachable variables and equations yields the concise final solved formula

$$\exists VZ\ X = g(X,Z) \wedge Z = f(V)\ .$$

5 Conclusion

We reconstructed Huet's tree equation solving algorithm for rational trees as a CHR solver for refined semantics with optimal almost-linear time complexity with improves on the quadratic complexity of [13]. To this end, we optimised the quadratic classic rational tree solver by combination with the almost-linear union-find solver. Our compact and highly concise code is shorter than implementations in other languages and even shorter than most formal expositions.

Moreover, we extended the CHR solver to solve existentially quantified conjunctions of non-flat equations in theory $\mathcal{T}$ in almost-linear time using the notion of reachability [13]. Our new definitions of solved, oriented, representative, and purged form are adapted to the union-find data-structure and yield

a more explicit answer, e.g. $\exists X\ Y = f(X)$ instead of the finally solved form $\exists X\ Y = f(X) \wedge X = Y$ from [13]. To the best of our knowledge, this is the first CHR solver for existentially quantified conjunctions of non-flat equations with *almost-linear time complexity*.

As unification is known to be inherently sequential (cf. [4]) future work aims to study the declarative concurrency of the CHR solver when using the parallel CHR union-find implementation [6].

We aim to extend the solver with entailment and disentailment as the basis of an algorithm for solving arbitrary first-order formulas involving equations and inequations in CHR.

References

1. F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.
2. G. J. Duck, P. J. Stuckey, M. J. García de la Banda, and C. Holzbaur. Extending arbitrary solvers with Constraint Handling Rules. In *PPDP'03*, pages 79–90. ACM Press, 2003.
3. G. J. Duck, P. J. Stuckey, M. J. García de la Banda, and C. Holzbaur. The refined operational semantics of Constraint Handling Rules. In *ICLP 2004*, volume 3132 of *LNCS*, pages 90–104. Springer, 2004.
4. C. Dwork, P. C. Kanellakis, and J. C. Mitchell. On the sequential nature of unification. *J. Logic Programming*, 1(1):35–50, 1984.
5. T. Frühwirth. Theory and Practice of Constraint Handling Rules. *J. Logic Programming*, 37(1–3):95–138, 1998.
6. T. Frühwirth. Parallelizing union-find in Constraint Handling Rules using confluence. In *ICLP 2005*, volume 3668 of *LNCS*, pages 113–127. Springer, 2005.
7. T. Frühwirth. Specialization of concurrent guarded multi-set transformation rules. In *LOPSTR 2004*, volume 3573 of *LNCS*, pages 133–148. Springer-Verlag, 2005.
8. T. Frühwirth and S. Abdennadher. *Essentials of Constraint Programming*. Springer, 2003.
9. G. P. Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *J. ACM*, 27(4):797–821, 1980.
10. K. Knight. Unification: a multidisciplinary survey. *ACM Comput. Surv.*, 21(1):93–124, 1989.
11. M. J. Maher. Complete axiomatizations of the algebras of finite, rational, and infinite trees. In *3rd Annual IEEE Symposium on Logic in Computer Science, LICS'88*, pages 348–357, Los Alamitos, CA, USA, 1988.
12. A. Martelli and G. Rossi. Efficient unification with infinite terms in logic programming. In *FGCS'84*, pages 202–209. ICOT, 1984.
13. M. Meister, K. Djelloul, and T. Frühwirth. Complexity of a CHR solver for existentially quantified conjunctions of equations over trees. In *Recent Advances in Constraints*, of *LNCS*. Springer, to appear.
14. T. Schrijvers and B. Demoen. The K.U.Leuven CHR system: implementation and application. In *CHR 2004, Selected Contributions*, volume 2004-01 of *Ulmer Informatik-Berichte*. Universität Ulm, Germany, 2004.
15. T. Schrijvers and T. Frühwirth. Optimal union-find in Constraint Handling Rules. *J. Theory and Practice of Logic Programming*, 6(1&2):213–224, 2006.
16. T. Schrijvers et al. The Constraint Handling Rules (CHR) web page, 2007. `http://www.cs.kuleuven.ac.be/~dtai/projects/CHR/`.
17. R. E. Tarjan and J. Van Leeuwen. Worst-case analysis of set union algorithms. *J. ACM*, 31(2):245–281, 1984.

Closures and Modules within Linear Logic Concurrent Constraint Programming

Rémy Haemmerlé, François Fages and Sylvain Soliman

INRIA Rocquencourt – France
FirstName.LastName@inria.fr

Abstract. There are two somewhat contradictory ways of looking at modules in a given programming language. On the one hand, module systems are largely independent of the particulars of programming languages. On the other hand, the module constructs may interfere with the programming constructs, and may be redundant with the other scope mechanisms of a specific programming language, such as closures for instance. There is therefore a need to unify the programming concepts that are similar, and retain a minimum number of essential constructs to avoid arbitrary programming choices. In this paper, we realize this aim in the framework of linear logic concurrent constraint programming (LCC) languages. We first show how declarations and closures can be internalized as agents in a variant of LCC with the bang operator of Linear Logic for which we provide precise operational and logical semantics. Then we show how a complete module system can be represented within LCC, and prove for it a code protection property. Finally we study the instanciation of this module system to constraint logic programming, and conclude on the generality of this scheme for programming languages with logical variables.

1 Introduction

Module systems are an essential feature of programming languages as they facilitate the re-use of existing code and the development of general purpose libraries. There are however two contradictory ways of looking at a module system. On the one hand, a module system is essentially independent of the particulars of a given programming language. "Modular" module systems have thus been designed and indeed adapted to different programming languages, see e.g. [13]. On the other hand, module constructs often interfere with the programming constructs and may be redundant with other scope mechanisms supported by a given programming language, such as closures for instance. There is therefore a need to unify the programming concepts that are similar in order to retain a minimum number of essential constructs and avoid arbitrary programming choices. In this paper, we realize this aim in the framework of linear logic concurrent constraint programming (LCC) languages.

The class of Concurrent Constraint (CC) programming languages was introduced in [16] as an elegant merge of constraint logic programming (CLP) and

concurrent logic programming. In the CC paradigm, CLP goals are concurrent agents communicating through a common store of constraints, each agent being able to post constraints to the store, and to synchronize by asking whether a guard constraint is entailed by the store. Research on the logical semantics of CC languages [6, 17] led to a simple solution in Girard's Linear Logic [7]. Through a straightforward translation of CC agents into intuitionistic LL (ILL) formulas, CC operational transitions indeed correspond to deductions in ILL, and completeness theorems hold for the observation of both the set of accessible stores, and the set of success stores [6].

Moreover, the soundness and completeness theorems still hold when considering constraint systems based on Linear Logic instead of classical logic, that constitutes the LCC framework. From a programming point of view, ILL constraint systems are a refinement of classical constraint systems allowing for the non-monotonic evolution of the constraint store, as advocated in [2], through the consumption of Linear Logic tokens by linear implication [6]. In LCC, constraint programming and imperative programming features are thus reconciled in a unified framework, and LCC is proposed as a kernel language for developing constraint programming libraries in a modular fashion in [8].

In this paper, we focus on a module system and a closure mechanism that can be naturally internalized in LCC. We first show in Section 2, that the linear tokens and the bang operator of LL can be used to internalize CC declarations and procedure calls as constraint posting and asking in LCC. A quite general notion of closure can then be encoded as a banged agent with an environment. The case of an empty environment corresponds to the usual CC declarations.

Then in Section 3, we develop a complete module system for LCC via a simple syntactical convention for encapsulating procedure declarations and calls. We prove a general property of code protection by showing that the implementation hiding is realized with the usual hiding operator for variables.

This module system is then illustrated in Section 4, by its instantiation to Constraint Logic Programming (CLP) languages, and by its relationship to the module system proposed in [9]. Its implementation is discussed here along the lines of its semantics in LCC, and is illustrated with examples of code hiding, closure programming and module parameterization in CLP.

Finally, we conclude on the generality of this approach for programming languages with logical variables.

Related Work

Concerning CC languages, the implementation of modules has not been much discussed up to now, being considered as an orthogonal issue. For instance, the MOZART-OZ language [15, 4] contains an *ad-hoc* module system allowing for separate compilation, but presented as an extra logical feature separated from the other programming constructs.

Concerning programming languages developed in Linear Logic using the Logic Programming paradigm, like for instance LO [1], Lolli [12] or Lygon [11],

it is worth noticing that persistent asks (which could be represented as implications under a ! in most of these languages) have not been considered, nor the direct encoding of dynamic clause assertions. On the other hand, the banged ask appears in the recent work of [14] on the expressiveness of linearity and persistence in process calculi for security. Here we use the full power provided by both persistent and non persistent inputs and both persistent and non-persistent outputs.

The internalization of declarations as agents proposed in this paper also goes somehow in the opposite direction to that of definition-based logics, as described for instance in [10]. Here, definitions are represented by first-class citizens as banged agents. This makes it possible to represent closures just by considering definitions that share variables with other agents.

2 Declarations as Agents

In this section, we give a presentation of LCC languages where the usual CC declarations are replaced by banged ask agents, called *persistent asks*. This construct actually generalizes declarations by providing closures where free variables in persistent asks represent the environment.

In this paper, a set of variables is denoted by capital letters X, Y, ... while a sequence of variables is denoted by $\boldsymbol{x}$, $\boldsymbol{y}$,... The set of free variables occurring in a formula A is denoted by $\mathcal{V}(A)$, $A[\boldsymbol{x}\backslash\boldsymbol{t}]$ denotes the formula A in which the free occurrences of variables $\boldsymbol{x}$ have been replaced by terms $\boldsymbol{t}$ (with the usual renaming of bound variables, avoiding variable clashes).

2.1 Linear Logic Constraint Systems

LCC languages essentially extend CC languages by considering constraint systems based on Girard's Linear Logic [7] instead of classical logic. From a programming point of view, this extension introduces state change and imperative features in constraint languages by allowing a non-monotonic evolution of the store of constraints [2]. In this section, we recall the definition of linear logic constraint systems as given in [6, 17].

Let $\mathcal{T}$ be the set of terms (noted t, s, ...) formed from a set V of variables and a set Σ_F of function symbols. An *atomic constraint* is a formula built from V, Σ_F and a set Σ_C of relation symbols, which does not contain $\top$, the neutral element of the additive linear conjunction. The *constraint language* is the least set containing all atomic constraints, closed by multiplicative conjunction ($\otimes$) existential quantification ($\exists$) and exponentiation (!).

Definition 1 (LL Constraint System). *A* linear constraint system *is a pair* $(\mathcal{C}, \Vdash_{\mathcal{C}})$ *where:*

- $\mathcal{C}$ *is a constraint language containing* $\mathbf{1}$ *the neutral element of the multiplicative conjunction.*

– $\Vdash_{\mathcal{C}}$ *is a subset of* $\mathcal{C} \times \mathcal{C}$ *which defines the non-logical axioms of the constraint system.*

The entailment relation $\vdash_{\mathcal{C}}$ *is the least subset of* $\mathcal{C} \times \mathcal{C}$ *containing* $\Vdash_{\mathcal{C}}$ *and closed by the rules of intuitionistic linear logic (the complete sequent calculus of ILL is given in appendix A) :*

$$c \vdash c \qquad \frac{\Gamma, c \vdash d \quad \Delta \vdash c}{\Gamma, \Delta \vdash d} \qquad \vdash \mathbf{1} \qquad \Gamma \vdash \top \qquad \frac{\Gamma \vdash c}{\Gamma, \mathbf{1} \vdash c} \qquad \Gamma, \mathbf{0} \vdash A$$

$$\frac{\Gamma \vdash c_1 \quad \Delta \vdash c_2}{\Gamma, \Delta \vdash c_1 \otimes c_2} \qquad \frac{\Gamma, c_1, c_2 \vdash c}{\Gamma, c_1 \otimes c_2 \vdash c} \qquad \frac{\Gamma \vdash c}{\Gamma \vdash \exists x\ c} \qquad \frac{\Gamma, c \vdash d}{\Gamma, \exists x\ c \vdash d}\ x \notin fv(\Gamma, d)$$

$$\frac{\Gamma, !d, !d \vdash c}{\Gamma, !d \vdash c} \qquad \frac{\Gamma \vdash c}{\Gamma, !d \vdash c} \qquad \frac{\Gamma, c \vdash d}{\Gamma, !c \vdash d} \qquad \frac{!\Gamma \vdash d}{!\Gamma \vdash !d}$$

In this setting, classical constraints are written under a bang !, while linear logic constraints without bang can be consumed by linear implication. In practice, the non classical constraints will be restricted to *linear tokens* which have no other axiom than the general axiom for equality :

$$p(x) \otimes !(x = y) \vdash p(y)$$

To this end, the vocabulary of predicate symbols Σ_C is partitioned into two sets Σ_D, Σ_P, where Σ_D contains the *classical constraints* with at least *true* ($\mathbf{1}$), *false* ($\mathbf{0}$) and $=$, and Σ_P contains the *linear token predicates*. The constraint languages built from Σ_D and Σ_P are noted $\mathcal{D}$ and $\mathcal{P}$ respectively.

Example 1. A typical LL constraint system is that of a combination of classical constraints, such as Herbrand terms, with linear tokens like $value(X, V)$ to encode imperative variables and assignment.

By an easy induction on proof trees we have :

Proposition 1. *Let* $c \in \mathcal{D}$ *and* $d \in \mathcal{P}$*. If* $c \vdash d \otimes \top$ *then* $c \vdash \mathbf{0}$*.*

The set of free variables occurring in the linear tokens of some constraint c is denoted by $\mathcal{V}_P(c)$. Formally, $\mathcal{V}_P(p(\boldsymbol{t})) = \mathcal{V}(\boldsymbol{t})$ if $p \in \Sigma_P$, $\mathcal{V}_P(p(\boldsymbol{t})) = \emptyset$ if $p \in \Sigma_D$, and the definition is extended to non-atomic constraints as usual.

2.2 Syntax of LCC

Given an LL constraint system $(\mathcal{C}, \Vdash_{\mathcal{C}})$, the syntax of LCC$(\mathcal{C}, \Vdash_{\mathcal{C}})$ agents is defined by the following grammar :

$A ::= A || A \mid \exists x.A \mid c \mid \forall \boldsymbol{x}(c \rightarrow A) \mid \forall \boldsymbol{x}(c \Rightarrow A)$

where c stands for any constraint in $\mathcal{C}$ and $\boldsymbol{x} \subset \mathcal{V}_P(c)$.

As usual $||$ stands for parallel composition, the $\exists$ operator hides variables in an agent, and the *tell* agent, written as a constraint, adds that constraint to the store. Two forms of ask agents are considered here : $\forall \boldsymbol{x}(c \rightarrow A)$ for the usual ask,

and $\forall \boldsymbol{x}(c \Rightarrow A)$ for the persistent ask. In both cases we impose furthermore that $\boldsymbol{x} \subset \mathcal{V}_P(c)$. This restriction limits the binding of variables by pattern matching to the variables occurring in linear tokens, and prevents the possible enumeration of all variables by ask agents.

The choice operator is defined here as an abbreviation :

$A + B = \exists x(choice(x) \,||\, choice(x) \Rightarrow A \,||\, choice(x) \Rightarrow B)$

Note that this abbreviation corresponds to the classical encoding of the non-deterministic choice in CLP with two clauses with the same head.

2.3 Operational Semantics

As in [6], the operational semantics of LCC is defined with a structural congruence and a transition relation defined over configurations. A *configuration* is a tuple $\langle X;\, c;\, \Gamma\rangle$ where X is a multi-set of variables, Γ a multi-set of agents and c a constraint, called *store*.

The structural congruence $\equiv$ is the least congruence satisfying the following two rules of *localisation* and the rule of *parallel composition*:

$$\frac{y \notin X \cup \mathcal{V}(\Gamma)}{\langle X;\, \exists y.c;\, \Gamma\rangle \equiv \langle X \cup \{y\};\, c;\, \Gamma\rangle} \qquad \frac{y \notin X \cup \mathcal{V}(c, \Gamma)}{\langle X;\, c;\, \exists y.A, \Gamma\rangle \equiv \langle X \cup \{y\};\, c;\, A, \Gamma\rangle}$$

$$\langle X;\, c;\, A \,||\, B, \Gamma\rangle \equiv \langle X;\, c;\, A, B, \Gamma\rangle$$

The transition relation $\longrightarrow$ is the least relation satisfying the following rules:

Tell $$\langle X;\, c;\, d, \Gamma\rangle \longrightarrow \langle X;\, c \otimes d;\, \Gamma\rangle$$

Ask $$\frac{c \vdash_{\mathcal{C}} d \otimes e}{\langle X;\, c;\, \forall \boldsymbol{z}(d \rightarrow A), \Gamma\rangle \longrightarrow \langle X;\, e;\, A[\boldsymbol{s}/\boldsymbol{z}], \Gamma\rangle}$$

Persistent ask $$\frac{c \vdash_{\mathcal{C}} d \otimes e}{\langle X;\, c;\, \forall \boldsymbol{z}(d \Rightarrow A), \Gamma\rangle \longrightarrow \langle X;\, e;\, A[\boldsymbol{s}/\boldsymbol{z}], \forall \boldsymbol{z}(d \Rightarrow A), \Gamma\rangle}$$

The transitive and reflexive closure of the transition relation is denoted by $\xrightarrow{*}$. The LCC transitions enjoy a general property of monotonicity that permits to define the operational semantics for arbitrary configurations from the observables of agents in an empty store of constraint.

Proposition 2 (Monotonicity). *If* $(X; c; \Gamma) \rightarrow (Y; d; \Delta)$ *then* $(X; c \otimes e; \Gamma, \Sigma) \rightarrow (Y; d \otimes e; \Delta, \Sigma)$ *for every constraint* e *and agents* Δ.

Definition 2 (Observables). *Let* A *be an LCC(*$\mathcal{C}$*) agent. We say that a constraint* $d \in \mathcal{C}$ *is an* accessible constraint *for* A *if there exists a derivation of the form* $\langle \emptyset;\, \mathbf{1};\, A\rangle \xrightarrow{*} \langle X;\, c;\, \Gamma\rangle$ *such that* $\exists X.c \vdash_{\mathcal{C}} d \otimes \top$. *Similarly,* d *is a* pseudo-success *for* A*, if* Γ *is a multi-set of persistent asks and* $\exists X.c \vdash_{\mathcal{C}} d$. d *is a* success *of* A*, if it is a pseudo-success for* A *such that* $\langle X;\, c;\, \Gamma\rangle \not\longrightarrow$.

Note that as a consequence of the rule for telling a constraint we have :

Proposition 3. *The two configurations* $\langle X; c; \Gamma, d\rangle$ *and* $\langle X; c \otimes d; \Gamma\rangle$ *have the same set of accessible constraints.*

Definition 3 (Operational Semantics).

- $\mathcal{O}^{const}(A)$ *is the set of accessible constraints for the agent* A.
- $\mathcal{O}^{\mathcal{D}const}(A) = \mathcal{O}^{const}(A) \cap \mathcal{D}$ *is the set of accessible* $\mathcal{D}$-constraints *for the agent* A.
- $\mathcal{O}^{succ}(A) =$ *is the set of successes for the agent* A.
- $\mathcal{O}^{Dsucc}(A) = \mathcal{O}^{succ}(A) \cap \mathcal{D}$ *is the set of* $\mathcal{D}$-successes *for the agent* A.

Example 2. The classical notion of *closure* can be recovered through a variable C not quantified in a persistent ask. Here $\forall X(apply(C, X) \Rightarrow min(X, minint) \otimes max(X, maxint))$ waits for its argument X to add constraints on it, defining it as an FD variable. From a functional perspective, a closure is basically a lambda expression with only one parameter, i.e. C is somehow equivalent to $(\lambda X.min(X, minint) \otimes max(X, maxint))$ and the agent $apply(C, X)$ to $C.X$.

This makes it possible to define iterators on data structures such as `forall` on lists, passing the closure as an argument as follows:

$\forall C.forall(C, [\,]) \Rightarrow true \;||$
$\forall H, T, C.forall(C, [H|T]) \Rightarrow apply(C, H) \otimes forall(T) \;||$

$\exists C.(\forall X(apply(C, X) \Rightarrow min(X, minint) \otimes max(X, maxint)) \;||\; forall(C, L))$

This illustrates how usual declarations are recovered through the use of persistent asks, and how free variables in persistent ask are used to provide the calling environment. Here the observable $\mathcal{O}^{Dsucc}$ corresponds to the expected application of our closure to a list.

2.4 Logical Semantics

In this section, we show how the usual ILL semantics of LCC [6] extend to persistent asks, with similar properties of soundness and completeness.

The translation of LCC agents into multiplicative ILL (MILL, see appendix A for the complete sequent calculus) is as follows :

$$c^\dagger = c \qquad (\exists x.c)^\dagger = \exists x.c^\dagger \qquad (A \;||\; B)^\dagger = A^\dagger \otimes B^\dagger$$
$$(\forall X(c \to A))^\dagger = \forall \overline{x}(c \multimap A^\dagger) \qquad (\forall X(c \Rightarrow A))^\dagger = !\forall \overline{x}(c^\dagger \multimap A^\dagger)$$

This translation extends to a multiset of agents Γ by $\{A_1, \ldots, A_n\}^\dagger = A_1^\dagger \otimes \cdots \otimes A_n\dagger$, and $\emptyset^\dagger = \mathbf{1}$. The translation of a configuration $\langle X; c; \Gamma\rangle$ is the formula $\langle X; c; \Gamma\rangle\,\dagger = \exists X.(c \otimes \Gamma)$.

As in [6], one obtains:

Theorem 1 (Soundness). *Let* $\langle X;\, c;\, \Gamma\rangle$ *and* $\langle Y;\, d;\, \Delta\rangle$ *be two LCC configurations.*

If $\langle X;\, c;\, \Gamma\rangle \equiv \langle Y;\, d;\, \Delta\rangle$ *then* $\langle X;\, c;\, \Gamma\rangle^\dagger \dashv\vdash_{\mathcal{C}} \langle Y;\, d;\, \Delta\rangle^\dagger$

If $\langle X;\, c;\, \Gamma\rangle \longrightarrow^* \langle Y;\, d;\, \Delta\rangle$ *then* $\langle X;\, c;\, \Gamma\rangle^\dagger \vdash_{\mathcal{C}} \langle Y;\, d;\, \Delta\rangle^\dagger$

Theorem 2 (Completeness[1]). *Let* $\{A_1, \ldots, A_n\}$ *be a multiset of agents and* c *a constraint, if* $A_1^\dagger, \ldots, A_n^\dagger \vdash_{\mathcal{C}} c$ *then there exists a derivation* $\langle \emptyset;\, \mathbf{1};\, A_1, \ldots, A_n\rangle \longrightarrow^* \langle X;\, d;\, \Gamma\rangle$ *where* $\exists X d \vdash_{\mathcal{C}} c$, Γ *is a multiset of* persistent asks.

Theorem 3 (Logical semantics of accessible constraints). *Let* A *be an LCC agent, we have:*

$$\mathcal{O}^{const}(A) = \{c \in \mathcal{C} \mid A^\dagger \vdash_{\mathcal{C}} c \otimes \top\} \qquad \mathcal{O}^{\mathcal{D}const}(A)) = \{d \in \mathcal{D} \mid A^\dagger \vdash_{\mathcal{C}} d \otimes \top\}$$

Proof. For the first inclusion, one simply uses the soundness theorem. For the second inclusion, the theorem 2 is used for the right introduction of the tensor connective in $c \otimes \top$ and one verifies that the property is preserved by the left introduction rules of ILL.

Because this translation of LCC agents uses the bang operator for the persistent asks, the set of final stores (and hence successes) cannot be characterized exactly for all constraints. Indeed the weakening rule for ! allows forgetting a formula corresponding to a persistent ask before consuming the entailed constraint of its guard. However, the property holds w.r.t. the observation of $\mathcal{D}$-successes.

Definition 4 ($\mathcal{P}$-persistent). *Let* $\mathcal{C}$ *be a constraint system partitionned into classical constraints* $\mathcal{D}$ *and linear tokens* $\mathcal{P}$. *An agent is* $\mathcal{P}$*-persistent if it contains no persistent asks with a guard belonging to* $\mathcal{D}$.

Theorem 4 (Logical semantics of $\mathcal{D}$-successes). *Let* A *be a* $\mathcal{P}$*-persistent LCC(*$\mathcal{C}$*) agent for which* $\mathbf{0}$ *is not an accessible constraint.*

$$\mathcal{O}^{Dsucc}(A) = \{d \in \mathcal{D} \mid A^\dagger \vdash_{\mathcal{C}} d\}$$

Proof. The first inclusion is immediate by soundness. For the other inclusion, by theorem 2, we get that for any constraint d of $\mathcal{D}$, s.t. $A^\dagger \vdash_{\mathcal{C}} d$, there exists a derivation $\langle \emptyset;\, \mathbf{1};\, A\rangle \longrightarrow^* \langle X;\, d';\, \Gamma\rangle$ where Γ is a multiset of persistent asks and $\exists X\ d' \vdash_{\mathcal{C}} d$. It is now enough to show that $\langle X;\, d';\, !\Gamma\rangle$ cannot be reduced. Let us suppose that there exists a persistent ask $\forall \boldsymbol{x}(c \Rightarrow B)$ in Γ such that $d' \vdash c \otimes e$ for some e in $\mathcal{C}$. We would then have $d' \vdash_{\mathcal{C}} c \otimes \top$, which thanks to proposition 1 contradicts the hypotheses since $c \notin \mathcal{D}$ (A and thus Γ is $\mathcal{P}$-persistent) qed. □

[1] Note to the reviewers: the proofs omitted in the text of this article are given in the appendix

3 Modules as Agents

3.1 Syntax

The declaration and closure mechanism provided by the persistent ask in LCC can be used to build a complete module system within LCC. In this approach, a module is named by a variable and the scope of module declarations thus depends on the scope of these variables[2]. Modular LCC adds the syntactical construct $x\{A\}$ for the localization of agent A in module x. Similarly telling a token constraint p must be localized with the syntax $x : p$ while the tell of a classical constraint d is not localized. The syntax of modular MLCC($\mathcal{C}$) *agents* is given by the following grammar:

$$A ::= x\{A\} \mid x\!:\!p \mid d \mid A \,||\, A \mid \exists x.A \mid \forall \boldsymbol{x}(c \to A) \mid \forall \boldsymbol{x}(c \Rightarrow A)$$

where p stands for a linear token constraint, d stands for a classical constraint and c stands for an arbitrary constraint.

Definition 5 (Translation in LCC). *The translation of an MLCC agent A in a module x is the LCC agent A^x defined by*

$g(\boldsymbol{s})^x = g(\boldsymbol{s}) \qquad p(\boldsymbol{s})^x = p(t, \boldsymbol{s}) \quad (c \otimes d)^x = c^x \otimes d^x \quad (\exists \boldsymbol{y}.c)^x = \exists \boldsymbol{y}.c^x \quad (!c)^x = !c^x$

$(\exists \boldsymbol{y}.A)^x = \exists \boldsymbol{y}.A^x \qquad (s\{A\})^x = A^s \qquad (s\!:\!c)^x = c^s \quad (A \,||\, B)^x = A^x \,||\, B^x$

$(\forall \boldsymbol{y}(c \to A))^x = \forall \boldsymbol{y}(c^x \to A^x) \qquad (\forall \boldsymbol{y}(c \Rightarrow A))^x = \forall \boldsymbol{y}(c^x \Rightarrow A^x)$

where $\boldsymbol{y} \cap \mathcal{V}(x) = \emptyset$ is supposed without loss of generality.

Example 3. A *List* module is defined here with an internal anonymous implementation of the linear *reverse* predicate:

$$\begin{array}{l} List\{\exists I \\ \quad \forall X, Y.reverse(X, Y) \Rightarrow I : reverse(X, [], Y) \;|| \\ \quad I\{ \\ \quad\quad \forall X.reverse([], X, X) \Rightarrow true \;|| \\ \quad\quad \forall X, Y, Z, T.reverse([X|Y], Z, T) \Rightarrow reverse(Y, [X|Z], T). \\ \quad \} \\ \} \end{array}$$

3.2 Code Protection

This section shows a general property of code protection. For this it is necessary to suppose that the non-logical axioms do not allow to make all variables equal. It is enough to enforce that for all syntactically distinct variables x and y by assuming that if $c \Vdash_{\mathcal{C}} x = y \otimes \top$ then $\{x, y\} \subset \mathcal{V}(c)$.

Definition 6. *The set $\mathcal{A}(X, c)$ of* accessible *variables from a constraint c and a set of variables X is defined inductively by:*

[2] It is worth noting that for the issue of separate compilation, modules should also be possibly named by constants making them visible by separate modules. This is considered in the next section.

$$\frac{x \in X \quad c \vdash_{\mathcal{C}} x = y \otimes \top}{y \in \mathcal{A}(X,c)} = \quad \frac{y \in \mathcal{A}(X,c)}{y \in \mathcal{A}(X, c \otimes d)} \otimes L \quad \frac{y \in \mathcal{A}(X,d)}{y \in \mathcal{A}(X, c \otimes d)} \otimes R \quad \frac{y \in \mathcal{A}(X,c)}{y \in \mathcal{A}(X, !c)} !$$

$$\frac{y \in \mathcal{A}(X \uplus z, c) \quad z \in \mathcal{A}(X,c)}{y \in \mathcal{A}(X,c)} \psi \quad \frac{y \in \mathcal{V}(\boldsymbol{s}) \quad x \subset X}{y \in \mathcal{A}(X, p(x,\boldsymbol{s}))} \Sigma_P \quad \frac{y \in \mathcal{A}(X,c) \setminus \{z\} \quad z \notin X}{y \in \mathcal{A}(X, \exists z.c)} \exists$$

For any multi-set of formulas Γ, $\mathcal{A}(\Gamma, c)$ will be used as a shortcut for $\mathcal{A}(\mathcal{V}(\Gamma), c)$. This notion of accessible variables serves to define the visibility of module predicates and prove a general property of code protection that shows that the code added inside a module cannot be observed from outside. This shows in particular that the implementation of a module is protected.

Lemma 1. *Let $c_1, \ldots, c_n$ and d be constraints and $\boldsymbol{x}$ be some variables not free in $c_1, \ldots c_n$ which appear in at least one linear tokens of d (i.e. $\boldsymbol{x} \subset \mathcal{V}_P(d)$ and $\boldsymbol{x} \cap \mathcal{V}(\Gamma) = \emptyset$). If $c_1, \ldots, c_n \vdash_{\mathcal{C}} d[\boldsymbol{x} \backslash \boldsymbol{t}]$ and $c_1, \ldots, c_n \nvdash_{\mathcal{C}} \mathbf{0}$ then $\mathcal{V}(\boldsymbol{t}) \subset \mathcal{A}(\mathcal{V}_P(d), (c_1, \ldots, c_n))$.*

Adding a new predicate to the implementation of a module cannot be observed whatever the external context.

Theorem 5 (Code Protection). *Let A, B, C and D be four MLCC agents, and y a variable, such that that any constraint posted by A and B are always of the form $x : A$. If A and B do not add any constraint on x, except those of the form $x : c$ nor any linear tokens of the form $p(\boldsymbol{t})$, then we have $\mathcal{O}^{\mathcal{D}const}(y\{\exists x.x\{A\} \mid\mid B\} \mid\mid C) = \mathcal{O}^{\mathcal{D}const}(y\{\exists x.x\{A \mid\mid \forall \boldsymbol{z}(p(\boldsymbol{z} \Rightarrow D)\} \mid\mid B\} \mid\mid C)$.*

Proof. One inclusion is obtained by the monotony of $\longrightarrow$.

We prove the other direction by induction on the transition $\langle \{x\}; \mathbf{1}; A^x, \forall \boldsymbol{z}(p(x,\boldsymbol{z}) \Rightarrow D^x), B^t, C\rangle \xrightarrow{*} \kappa = \langle X; c; \Delta, \forall \boldsymbol{z}(p(x,\boldsymbol{z}) \Rightarrow D^x), \Gamma, \Sigma\rangle$ where the Δ, Γ and Σ are the respective reductions of A^x, B^t and C. $c \nvdash_{\mathcal{C}} p(x,\boldsymbol{t}) \otimes \top$ and $x \notin \mathcal{A}(\Gamma, c)$

The base case is trivial. Now we are interested first, by structural congruence : let us suppose that $\kappa \equiv \langle X'; c'; \Delta', \forall \boldsymbol{z}(p(x,\boldsymbol{z}) \Rightarrow D^x), \Gamma', \Sigma'\rangle$:

- the case of *parallel composition* is trivial.
- for the *store hiding* : we know that the variable handled z is different from x since x is not free in the agent part. In such a case just notice that, we have, for any d, $x \in \mathcal{A}(Y,d)$ iff $x \in \mathcal{A}(Y, \exists z.d)$ and $d \nvdash_{\mathcal{C}} p(x,\boldsymbol{t}) \otimes \top$ iff $\exists z.d \nvdash_{\mathcal{C}} p(x,\boldsymbol{t}) \otimes \top$.
- the case of *agent hiding* is trivial.

Now we suppose that $\kappa \longrightarrow \kappa' = \langle X'; c'; \Delta', \forall \boldsymbol{z}(p(\boldsymbol{z}) \Rightarrow D^x), \Gamma', \Sigma'\rangle$.

- For the *tell* : $\kappa' \langle X; c \otimes d; \Delta', \forall \boldsymbol{z}(p(\boldsymbol{z}) \Rightarrow D^x), \Gamma', \Sigma'\rangle$. If the *tell* occurs in Δ, d is of the form e^x with $x \in \mathcal{V}(e)$. Notice that if $x \notin \mathcal{V}(e)$ and $x \notin \mathcal{A}(Y,c)$ then $x \notin \mathcal{A}(Y, c \otimes e^x)$. Moreover it is clear that if $x \notin \mathcal{V}(e)$ if $c \otimes e^x \vdash_{\mathcal{C}} x = y \otimes \top$

then $c \vdash_{\mathcal{C}} x{=}y \otimes \top$. Hence, since e^x does not contain any token of the form $p(x, \boldsymbol{t})$, if $c \otimes e^x \vdash_{\mathcal{C}} p(x, \boldsymbol{t}) \otimes \top$ then $c \vdash_{\mathcal{C}} p(x, \boldsymbol{t}) \otimes \top$.
The case, where the *tell* occurs in Γ, is similar.
If the *tell* occurs in Σ. Notice that $av((\Gamma', d), c) \subset av(\Gamma', c \otimes d)$. Moreover since x is not free in Γ) and then in d, we have clearly if $c \nvdash_{\mathcal{C}} p(x, \boldsymbol{t}) \otimes \top$ then $c \otimes d \nvdash_{\mathcal{C}} p(x, \boldsymbol{t}) \otimes \top$.

- For the *persistent ask* : Of course if $c \nvdash p(x, \boldsymbol{t}) \otimes \top$ then $c' \nvdash p(x, \boldsymbol{t}) \otimes \top$. We just need to prove that $x \notin \mathcal{A}(\Sigma', c')$. The only interesting case if where the *tell* occurs in Σ then $\Sigma' = \Sigma, D[\boldsymbol{y}\backslash\boldsymbol{s}]$. Since by induction hypothesis, $x \notin \mathcal{A}(\Sigma, c')$, we have $x \notin \mathcal{A}(d, c')$, by using previous lemma we infer that $x \notin \mathcal{V}(\boldsymbol{s})$. Using definition of accessible variables it is clear that $x \notin \Sigma, D[\boldsymbol{y}\backslash\boldsymbol{s}]$.
- the case of *ask* is similar.

Note that the code protection property defined here is implied by the called module code protection presented in [9] for Prolog module systems. It is violated by the same module systems for Prolog as in [9].

4 A Module System for CLP

The MLCC scheme presented above can be instantiated into a powerful yet simple module system for Constraint Logic Programming (CLP) languages. This module system is similar to the one proposed for CLP in [9] for which we thus provide here a logical semantics in linear logic, and an implementation with continuations in the line of its semantics in LCC. This resulting language, called mCLP for modular CLP, has been implemented in a "proof of concept" prototype[3].

4.1 mCLP Syntax

We shall adopt for mCLP a pragmatic syntax close to that of classical CLP systems. The `typewriter` font is used for programs, where, as in classical Prolog programs, the identifiers beginning with a capital letter represent variables. The syntax defined by the following grammar distinguishes declarations from goals as usual:

```
G ::= module(T,E){D} | T:p(S1,...,Sn) |
 p(S1,...,Sn) | c(S1,...,Sn) | G,G | G;G

D ::= p(S1,...,Sn) :- G.D | p(S1,...,Sn).D |
 :- G.D | ϵ
```

where `T` is a term, `E` a list of variables, $\mathtt{S}_1\mathtt{,...,S}_n$ a sequence of terms, `c` a constraint of $\mathcal{C}$ and `p` a predicate construct using the predicate symbols alphabet Σ_P.

[3] The prototype implementation of mCLP is available for download at the following address: `http://contraintes.inria.fr/~haemmerl/pub/mclp.tgz`

An mCLP declaration is either a clause, a fact or a goal of the form `:- G.` executed at the initialization of the module.

Besides the usual conjunction, disjunction and constraint posting goals, the goal `module(T, E){D}` denotes the *instantiation* of a module `T` with the *implementation* `D` and the *environment* `E`. This environment is simply a list of *global variables* whose scope is the entire module clauses. If `T` is a free variable, the resulting module is *anonymous*, whereas if `T` is an atom (or a compound term), it is a *named* module, as proved useful for separate compilation.

The goal $\mathtt{T{:}p(S_1,\ \ldots,\ S_n)}$ denotes the *external call* of the predicate `p/n` defined in the module `T`, which is distinguished from the *local call*, noted $\widehat{\mathtt{p}}(\mathtt{S}_1, \ldots, \mathtt{S}_n)$, of the predicate `p/n` defined in the current module.

4.2 Interpretation of mCLP into MLCC

Classical clauses are interpreted by *persistent asks* waiting for the linear token that represents the procedure call. The module environment provides a new feature allowing for global variables in a module. Formally, the interpretation of mCLP goals and declaration in MLCC is defined by $[\![\mathtt{G}]\!]^{\mathtt{T}}$ and $[\![\mathtt{D}]\!]^{\mathtt{T}}_{\mathtt{E}}$ where `T` is the current module and `E` the current environment:

$$[\![\mathtt{G}_1,\mathtt{G}_2]\!]^{\mathtt{T}} = [\![\mathtt{G}_1]\!]^{\mathtt{T}} \mid\mid [\![\mathtt{G}_2]\!]^{\mathtt{T}} \qquad [\![\mathtt{P}]\!]^{\mathtt{T}} = \mathtt{T{:}P} \qquad [\![\mathtt{S{:}P}]\!]^{\mathtt{T}} = \mathtt{S{:}P}$$
$$[\![\mathtt{G}_1;\mathtt{G}_2]\!]^{\mathtt{T}} = [\![\mathtt{G}_1]\!]^{\mathtt{T}} + [\![\mathtt{G}_2]\!]^{\mathtt{T}} \qquad [\![\mathtt{C}]\!]^{\mathtt{T}} = \mathtt{T{:}(!C)} \qquad [\![\mathtt{module(S,E)\{D\}}]\!]^{\mathtt{T}} = \mathtt{S}\{[\![\mathtt{D}]\!]^{\mathtt{S}}_{\mathtt{E}}\}$$

$$[\![\ \mathtt{:-\,G.D}]\!]^{\mathtt{T}}_{\mathtt{E}} = \exists\overline{\mathtt{Y}}[\![\mathtt{G}]\!]^{\mathtt{S}} \mid\mid [\![\mathtt{D}]\!]^{\mathtt{T}}_{\mathtt{E}}$$
$$[\![\mathtt{p(t).D}]\!]^{\mathtt{T}}_{\mathtt{E}} = \forall\mathtt{X}(\mathtt{p(X)} \Rightarrow \exists\overline{\mathtt{Y}}[\![\mathtt{X{=}t}]\!]^{\mathtt{S}}) \mid\mid [\![\mathtt{D}]\!]^{\mathtt{T}}_{\mathtt{E}}$$
$$[\![\mathtt{p(t)\ :-\,G.D}]\!]^{\mathtt{T}}_{\mathtt{E}} = \forall\mathtt{X}(\mathtt{p(X)} \Rightarrow \exists\overline{\mathtt{Y}}[\![\mathtt{X} = \mathtt{t}, \mathtt{G}]\!]^{\mathtt{S}}) \mid\mid [\![\mathtt{D}]\!]^{\mathtt{T}}_{\mathtt{E}}$$

where `X` is a set of fresh variables and $\overline{\mathtt{Y}} = \mathcal{V}(\mathtt{t},\mathtt{G}) \setminus \mathtt{E}$.

This translation is supposed to work on the linear constraint system $(\mathcal{CP}, \Vdash_{\mathcal{CP}})$ such that $\Vdash_{\mathcal{CP}}$ is the smallest set respecting the following conditions:

- If $(\mathtt{C} \Vdash_{\mathcal{C}^\circ} \mathtt{C})$ then $(\mathtt{C} \Vdash_{\mathcal{CP}} \mathtt{C})$.
- For any predicate symbol `p` $(\mathtt{p(X)}, \mathtt{X{=}Y} \Vdash_{\mathcal{CP}} \mathtt{p(Y)})$.

where $\Vdash_{\mathcal{C}^\circ}$ is the translation of the non-logical axioms of the classical constraint system $\mathcal{C}$ into linear logic (using for example the well know Girard's translation classical logical into linear logic [7]).

Notice that all the $[\![A]\!]^{\mathtt{T}}_{\mathtt{E}}$ are $\mathcal{P}$-persistent (see Def. 4), therefore all results of previous Section can be applied to mCLP programs.

4.3 Global Variables

Module environments introduce *global* variables, i.e. variables shared among the different clauses of the module. This construct can be used for instance to avoid passing an argument to numerous module predicates. However, these variables are still usual, backtrackable, logic variables.

The following code illustrates the use of a global variable `Depth` to implement a Prolog meta-interpreter with a fair search strategy proceeding by iterative deepening [18].

The predicate *clause* looks for clause definitions [5]; the predicate `for(I, Begin, End)` produces a choice point where `I` will be assigned any of the integer values between `Begin` and `End` (see for instance [3]).

Example 4. (Iterative Deepening):

```
:-module(iter_deep, [Depth]){

   solve(G):-
      for(Depth,1,1000),
         write('Depth: '),
         write(Depth),
         nl,
         iterative_deepening(G,0).

   iterative_deepening(_,I) :-
      I >= Depth,
      !,
      fail.

   iterative_deepening(((A,B)),I) :-
      !,
      iterative_deepening(A,I),
      iterative_deepening(B,I).

   iterative_deepening(A,_) :-
      clause((A:-true)),
      !.

   iterative_deepening(A,I) :-
      clause((A:-B)),
      J is I+1,
      iterative_deepening(B,J).
}
```

4.4 Code Hiding

As above, one can use an environment to make a variable *global* to a module, but this time, this variable will be used to keep an anonymous inside module hidden from the outside. Since the *name* of the inside module is this variable, only known to the clauses inside the module definition, the corresponding implementation is accessible only from the clauses of the outside module.

This is illustrated in the following program that provides the `sort` predicate and hides the implementation `quicksort` predicate.

Example 5. (Quicksort):

```
:- module(sort, [Impl]){

   sort(List,SortedList):-
      Impl:quicksort(List, SortedList).

   :- module(Impl, []){

      quicksort([],[]).
      quicksort([X|Tail], Sorted) :-
         split(X, Tail, Small, Big),
         quicksort(Small, SortedSmall),
         quicksort(Big, SortedBig),
         list:append(SortedSmall,
            [X|SortedBig], Sorted).

      split(X, [],[],[]).
      split(X, [Y|Tail], [Y|Small], Big) :-
         X<Y, !,
         split(X, Tail, Small, Big).
      split(X, [Y|Tail], Small, [Y|Big]) :-
         split(X, Tail, Small, Big).
   }.
}.
```

The code protection property 3.2 ensures that no call to the `quicksort` predicate is possible outside the `sort` predicate.

The execution of the goal

```
? L=[1, 2/3, 5, 4/3, 1/2, 2/7], sort:sort(L, L1), print(L1), nl.
```

prints on screen the sorted list `[2/7,1/2,2/3,1,4/3,5]`.

4.5 Closures

The classical notion of *closure* can be recovered through the definition of modules with a predicate `apply/1` waiting for the argument to apply the persitant ask (corresponding to the clauses of `apply/1`).

This makes it possible to define iterators on data structures such as `forall` or `exists` on lists, passing the closure as an argument as follows:

Example 6. :

```
:- module(iterator, []){

   forall([], _).
   forall([H|T], C) :- C:apply(H), forall(T, C).

   exists([H|_], C) :- C:apply(H).
   exists([_|T], C) :- exists(T, C).

}.
```

The usual `domain/3` (or `fd_domain/3`) built-in predicate of finite domain constraint solvers, can be implemented using the list iterator on its arguments:

```
fd_domain(Vars, Min, Max):-
   module(Cl , [Min, Max]){
      apply(X) :- Min=<X , X=<Max.
   },
   ( list(Vars) -> iterator:forall(Vars, Cl) ;
     var(Vars) -> Cl:apply(Vars) ).
```

4.6 Module Parameterization

Parameterized modules greatly enhance the programmer capabilities to re-use code by making its module implementation depend on other modules.

Combining the idea of using the environment to parameterize a closure, and the code hiding features demonstrated above, one can obtain a module with a hidden implementation, parameterized from outside.

The following example shows how to parameterize the previous *sort* module by creating a `generic_sort/2` predicate that dynamically creates a sorting module (its first argument) using the comparison predicate given as second argument.

Example 7. (Parameterized quicksort):

```
:- module(sort, []){

   generic_sort(Sort, Order) :-
      module(Sort, [Order, Impl]){

         sort(List, SortedList):-
            Impl:quicksort(List, SortedList).

         :- module(Impl, [Order]){

            quicksort([],[]).
            quicksort([X|Tail], Sorted) :-
               split(X, Tail, Small, Big),
               quicksort(Small, SortedSmall),
               quicksort(Big, SortedBig),
               list:append(SortedSmall,
                  [X|SortedBig], Sorted).

            split(X, [],[],[]).
            split(X, [Y|Tail], [Y|Small], Big) :-
               Order:(X >= Y), !,
               split(X, Tail, Small, Big).
            split(X, [Y|Tail], Small, [Y|Big]) :-
               split(X, Tail, Small, Big).
         }.
      }.
}.
```

By supposing there exist two modules `math` and `term` implementing the respective ISO predicates `>=` and `@>=`, the execution of the following goal prints on screen the two lists `[2/7,1/2,2/3,1,4/3,5]` and `[1,5,1/2,2/3,2/7,4/3]` :

```
L=[1, 2/3, 5, 4/3, 1/2, 2/7],
sort:factory(Sort1, math), Sort1:sort(L, L1), print(L1), nl,
module(OrderLex, []){ X >= Y:- term:(X @>= Y) },
sort:factory(Sort2, OrderLex), Sort2:sort(L, L2) print(L2), nl.
```

5 Conclusion

We have shown that a powerful module system for linear concurrent constraint programming (LCC) languages can be internalized into LCC, by representing declarations by persistent asks, referencing modules by variables and thus benefiting from implementation hiding through the usual hiding operator for variables. We have presented the operational semantics of MLCC programs, showing a code protection property, and proving the equivalence with the logical semantics in linear logic for the observation of stores and successes.

These results have been illustrated with an instantiation of the MLCC scheme to constraint logic programs, leading to a simple yet powerful module system for CLP similar to [9], supporting code hiding, closures and module parameterization, and with a logical semantics in linear logic.

We believe that this approach to internalizing a module system within a programming language is of a quite general scope for programming languages with logical variables, as well as its implementation with a continuation mechanism.

References

1. J.-M. Andreoli and R. Pareschi. Linear objects: Logical processes with built-in inheritance. *New Generation Computing*, 9:445–473, 1991.
2. E. Best, F. S. de Boer, and C. Palamidessi. Concurrent constraint programming with information removal. In *Proceedings of Coordination*, Lecture Notes in Computer Science. Springer-Verlag, 1997.
3. D. Diaz. *GNU Prolog user's manual*, 1999–2003.
4. D. Duchier, L. Kornstaedt, C. Schulte, and G. Smolka. A higher-order module discipline with separate compilation, dynamic linking, and pickling. draft, 1998.
5. P. D. A. Ed-Dbali and L. Cervoni. *Prolog: The Standard*. Springer-Verlag, New York, 1996.
6. F. Fages, P. Ruet, and S. Soliman. Linear concurrent constraint programming: operational and phase semantics. *Information and Computation*, 165(1):14–41, Feb. 2001.
7. J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50(1), 1987.
8. R. Haemmerlé. SiLCC is linear concurrent constraint programming (doctoral consortium). In M. Gabbrielli and G. Gupta, editors, *Proceedings of International Conference on Logic Programming ICLP 2005*, volume 3668 of *Lecture Notes in Computer Science*, pages 448–449. Springer-Verlag, 2005.

9. R. Haemmerlé and F. Fages. Modules for Prolog revisited. In *Proceedings of International Conference on Logic Programming ICLP 2006*, number 4079 in Lecture Notes in Computer Science, pages 41–55. Springer-Verlag, 2006.
10. L. Hallnäs. A proof-theoretic approach to logic programming. ii. programs as definitions. *Journal of Logic and Computation*, 1(5):635–660, Oct. 1991.
11. J. Harland, D. J. Pym, and M. Winikoff. Programming in lygon: An overview. In *Proceedings of the Fifth International Conference on Algebraic Methodology and Software Technology, Munich*, pages 391–405, July 1996.
12. J. S. Hodas and D. Miller. Logic programming in a fragment of intuitionistic linear logic. *Information and Computation*, 110(2):327–365, 1994.
13. X. Leroy. A modular module system. *Journal of Functional Programming*, 10(3):269–303, 2000.
14. C. Palamidessi, V. A. Saraswat, F. D. Valencia, and B. Victor. On the expressiveness of linearity vs persistence in the asychronous pi-calculus. In *LICS'06: Proceedings of the 21th Annual IEEE Symposium on Logic In Computer Science*, pages 59–68, 2006.
15. P. V. Roy, P. Brand, D. Duchier, S. Haridi, M. Henz, and C. Schulte. Logic programming in the context of multiparadigm programming: the Oz experience. *Theory and Practice of Logic Programming*, 3(6):715–763, Nov. 2003.
16. V. A. Saraswat. *Concurrent constraint programming.* ACM Doctoral Dissertation Awards. MIT Press, 1993.
17. V. A. Saraswat and P. Lincoln. Higher-order linear concurrent constraint programming. Technical report, Xerox Parc, 1992.
18. M. E. Stickel. A prolog technology theorem prover: implementation by an extended prolog compiler. *Journal of Automated Reasoning*, 44:353–380, 1988.

A Intuitionistic Linear Logic

The *intuitionistic formulae* are built from atoms p, q, ... with the multiplicative connectives $\otimes$ (tensor) and $\multimap$ (linear implication), the additive connectives & *(with)* and $\oplus$ *(plus)* the exponential connective ! (bang), and the universal $\forall$ and existential $\exists$ quantifiers.

The *intuitionistic sequents* are of the form $\Gamma \vdash A$, where A is a formula and Γ is a multi-set of formulae.

The sequent calculus is given by the following rules, where the basic idea is that the disappearance of the weakening rule makes the conjunction $\otimes$ count the occurrences of formulae, and the implication $\multimap$ consumes its premises [7]:

Axiom - Cut

$$A \vdash A \qquad \frac{\Gamma \vdash A \quad \Delta, A \vdash B}{\Delta, \Gamma \vdash B}$$

Constants

$$\frac{\Gamma \vdash A}{\Gamma, \mathbf{1} \vdash A} \quad \vdash \mathbf{1} \quad \Gamma \vdash \top$$

$$\bot \vdash \quad \frac{\Gamma \vdash}{\Gamma \vdash \bot} \quad \Gamma, \mathbf{0} \vdash A$$

Multiplicatives

$$\frac{\Gamma, A, B \vdash C}{\Gamma, A \otimes B \vdash C} \qquad \frac{\Gamma \vdash A \quad \Delta, B \vdash C}{\Delta, \Gamma, A \multimap B \vdash C}$$

$$\frac{\Gamma \vdash A \quad \Delta \vdash B}{\Gamma, \Delta \vdash A \otimes B} \qquad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \multimap B}$$

Additives

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \oplus B} \qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \oplus B}$$

$$\frac{\Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma, A \oplus B \vdash C} \qquad \frac{\Gamma, A \vdash C}{\Gamma, A \,\&\, B \vdash C}$$

$$\frac{\Gamma, B \vdash C}{\Gamma, A \,\&\, B \vdash C} \qquad \frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \,\&\, B}$$

Bang

$$\frac{\Gamma, A \vdash B}{\Gamma, !A \vdash B} \qquad \frac{!\Gamma \vdash A}{!\Gamma \vdash !A}$$

$$\frac{\Gamma, !A, !A \vdash B}{\Gamma, !A \vdash B} \qquad \frac{\Gamma \vdash B}{\Gamma, !A \vdash B}$$

Quantifiers

$$\frac{\Gamma, A[t/x] \vdash B}{\Gamma, \forall x A \vdash B} \qquad \frac{\Gamma \vdash A}{\Gamma \vdash \forall x A}\ x \notin \mathcal{V}(\Gamma)$$

$$\frac{\Gamma, A \vdash B}{\Gamma, \exists x A \vdash B}\ x \notin \mathcal{V}(y\Gamma, B) \qquad \frac{\Gamma \vdash A[t/x]}{\Gamma \vdash \exists x A}$$

B Proofs of Logical Semantics

B.1 Logical Semantics

Proof (Theorem 2). In the following proof $!\Delta$ and $!\Gamma$ will be notations for multi-sets of persistent asks. We will suppose w.l.o.g. that the variables in X are free in c. The result is proved by induction on the proof π of the sequent $A_1^\dagger, \ldots, A_n^\dagger \vdash_{\mathcal{C}} c$ where the A_i's are agents and c is a constraint.

First note that the induction is meaningful. Indeed the only cuts that cannot be eliminated in a proof bear on non-logical axioms, so they are of one of the following forms

$$\frac{\Gamma \vdash_{\mathcal{C}} d \quad \overline{d \vdash_{\mathcal{C}} c}}{\Gamma \vdash_{\mathcal{C}} c} \qquad \frac{\overline{d \vdash_{\mathcal{C}} e} \quad \Gamma, e \vdash_{\mathcal{C}} c}{\Gamma, d \vdash_{\mathcal{C}} c}$$

The introduction (from the bottom to the top) of one of these rules introduces a sequent where the right-hand side is a constraint. The same is true of the left introduction of $\multimap$. □

– π is an axiom: $c \vdash_{\mathcal{C}} d$. Just remark that $(\emptyset; \mathbf{1}; c) \longrightarrow (\emptyset; c; \emptyset)$.
– π ends with a cut :

$$\frac{\Gamma^\dagger \vdash_{\mathcal{C}} c \quad \overline{c \vdash_{\mathcal{C}} d}}{\Gamma^\dagger \vdash_{\mathcal{C}} d} \qquad \text{or} \qquad \frac{\overline{c \vdash_{\mathcal{C}} c'} \quad \Gamma^\dagger, c' \vdash_{\mathcal{C}} d}{\Gamma^\dagger, c \vdash_{\mathcal{C}} d}$$

The first case is immediate. For the second one, we know, by induction hypothesis, that $(\emptyset; \mathbf{1}; \Gamma, c') \xrightarrow{*} (X; d'; !\Delta)$ such that $\exists X d' \vdash_{\mathcal{C}} d$ and $X \cap \mathcal{V}(d) = \emptyset$. In this case, remark that the application of the *tell* rule which reduces the agent c' can be applied to the agent c.
– π ends with a left introduction of $\mathbf{1}$:

$$\frac{\Gamma^\dagger \vdash_{\mathcal{C}} c}{\Gamma^\dagger, \mathbf{1} \vdash_{\mathcal{C}} c}$$

First one remarks that $(\emptyset; \mathbf{1}; \mathbf{1}, \Gamma) \longrightarrow (\emptyset; \mathbf{1}; \Gamma)$. By induction hypothesis, we know moreover that $(\emptyset; \mathbf{1}; \Gamma) \xrightarrow{*} (\emptyset; d; !\Delta)$ such that $\exists X d \vdash_{\mathcal{C}} c$ and $X \cap \mathcal{V}(c) = \emptyset$. Hence $(\emptyset; \mathbf{1}; \mathbf{1}, \Gamma) \xrightarrow{*} (\emptyset; d; !\Delta)$.
– π ends with a right introduction of $\mathbf{1}$: just use the reflexivity of $\xrightarrow{*}$.
– π ends with a left introduction of $\mathbf{0}$: $\Gamma^\dagger, \mathbf{0} \vdash_{\mathcal{C}} c$
Notice that $(\emptyset; \mathbf{1}; \mathbf{0}, \Gamma) \longrightarrow (\emptyset; \mathbf{0}; \Gamma)$ and of course $\mathbf{0} \vdash_{\mathcal{C}} c$.
– π ends with a left introduction of $\otimes$:

$$\frac{\Gamma^\dagger, A, B \vdash_{\mathcal{C}} c}{\Gamma^\dagger, A \otimes B \vdash_{\mathcal{C}} c}$$

There are two subcases :
- $A \otimes B$ is the translation of a parallel composition of two agents. In such a case just use the *parallel composition* rule.
- $A \otimes B$ is the translation of a constraint of the form $c \otimes d$. In such case just notice that the two configurations $(\emptyset; \mathbf{1}; c, d, \Gamma)$ and $(\emptyset; \mathbf{1}; c \otimes d, \Gamma)$ have the same pseudo-successes.

– π ends with a right introduction of $\otimes$:

$$\frac{\Gamma^\dagger \vdash_{\mathcal{C}} c \quad \Delta \vdash_{\mathcal{C}} d}{\Gamma^\dagger, \Delta^\dagger \vdash_{\mathcal{C}} c \otimes d}$$

By induction hypothesis we know that $(\emptyset; \mathbf{1}; \Gamma) \xrightarrow{*} (X; c'; !\Gamma')$ and $(\emptyset; \mathbf{1}; \Delta) \xrightarrow{*} (Y; d'; !\Delta')$ with $\exists X c' \vdash_{\mathcal{C}} c$, $\exists Y d' \vdash_{\mathcal{C}} d$ and $X \cap \mathcal{V}(c) = \emptyset$ and $Y \cap \mathcal{V}(d) = \emptyset$. Thanks to the monotonicity of $\xrightarrow{*}$ we infer that $(\emptyset; \mathbf{1}; \Gamma, \Delta) \xrightarrow{*} (X; c'; \Gamma', \Delta) \xrightarrow{*} (X; c' \otimes d'; \Gamma', \Delta')$. To conclude it is enough to notice that for $X \cap \mathcal{V}(c) = \emptyset$ and $Y \cap \mathcal{V}(d) = \emptyset$ if $\exists X c' \vdash_{\mathcal{C}} c$ and $\exists X d' \vdash_{\mathcal{C}} d$ then $\exists XY(c' \otimes d') \vdash c \otimes d$.
– π ends with a left introduction of $\exists$:

$$\frac{\Gamma^\dagger, A\dagger \vdash_{\mathcal{C}} c}{\Gamma^\dagger, \exists x.A^\dagger \vdash_{\mathcal{C}} c} x \notin \mathcal{V}(\Gamma, c)$$

By induction hypothesis we know that $(\emptyset; \mathbf{1}; A, \Gamma) \xrightarrow{*} (X; d; !\Delta))$ with $\exists X.d \vdash_{\mathcal{C}} c$ and w.l.o.g. $X \cap \mathcal{V}(c) = \emptyset$. Then by monotonicity of $\xrightarrow{*}$, we have $(x; \mathbf{1}; A, \Gamma) \xrightarrow{*} (X', x; d[X \backslash X']; !\Delta[X \backslash X']))$ with, w.l.o.g. , $X' \cap \mathcal{V}(c) = \emptyset$. Because $x \notin \mathcal{V}(\Gamma)$, we can infer that $(\emptyset; \mathbf{1}; \exists x A, \Gamma) \equiv (x; \mathbf{1}; A, \Gamma)$. We conclude by noticing that if $\exists X d \vdash_{\mathcal{C}} c$ and $X' \cap \mathcal{V}(c) = \emptyset$ then $\exists X'.d[X \backslash X'] \vdash_{\mathcal{C}} c$.

– π ends with right introduction of $\exists$: immediate.
– π ends with a left introduction of $\multimap$:

$$\frac{\Gamma \vdash d \quad \Delta, A \vdash c}{\Gamma, \Delta, \forall d \multimap A \vdash_{\mathcal{C}} c}$$

By induction hypothesis we know that $(\emptyset; \mathbf{1}; \Gamma) \xrightarrow{*} (X; d'; !\Gamma')$ and $(\emptyset; \mathbf{1}; \Delta) \xrightarrow{*} (Y; c'; !\Delta')$. Thanks to the α-equivalence, we suppose w.l.o.g. that $X \cap Y = \emptyset$. We simply use the monotonicity of $\xrightarrow{*}$ and the *ask* rule to conclude that $\langle \emptyset;\ \mathbf{1};\ \Gamma, \Delta, d \to A \rangle \xrightarrow{*} \langle X;\ d';\ !\Gamma', \Delta, d \to A \rangle \longrightarrow \langle X;\ \mathbf{1};\ !\Gamma', \Delta, A \rangle \xrightarrow{*} \langle X, Y;\ c';\ !\Gamma', !\Delta', A \rangle$.
– ends with a left introduction of $\forall$:

$$\frac{\Gamma, A[\boldsymbol{x}\backslash\boldsymbol{t}] \vdash c}{\Gamma, \forall \boldsymbol{x} A \vdash c}$$

A is necessarily a translation of an *ask* $d \to B$. We verify easily that if $d[\boldsymbol{x}\backslash\boldsymbol{t}] \to B[\boldsymbol{x}\backslash\boldsymbol{t}]$ can be reduced then $\forall \boldsymbol{x}(d \to B)$ can be reduced too.
– π ends with a dereliction : There are two subcases :

$$\frac{\Gamma^\dagger, d^\dagger \vdash_{\mathcal{C}} c}{\Gamma^\dagger, !d^\dagger \vdash_{\mathcal{C}} c} \qquad \text{or} \qquad \frac{\Gamma^\dagger, \forall \boldsymbol{z}.(d^\dagger \multimap A^\dagger) \vdash_{\mathcal{C}} c}{\Gamma^\dagger, !\forall \boldsymbol{z}.(d^\dagger \multimap A^\dagger) \vdash_{\mathcal{C}} c}$$

The first case is obvious, just recall that $!d \vdash d$. For the second one, we know by induction hypothesis $\langle \emptyset;\ \mathbf{1};\ \Gamma, \forall \boldsymbol{x}(d \to A) \rangle \xrightarrow{*} \langle X;\ c';\ !\Gamma' \rangle$ with $\exists X.c' \vdash_{\mathcal{C}} c$ and $X \cap \mathcal{V}(c) = \emptyset$. Hence in this derivation replacing the application of the *ask* rule with reduces the agent $\forall \boldsymbol{x}(d \to A)$ by the *persistent ask* rule we obtain the derivation $\langle \emptyset;\ \mathbf{1};\ \Gamma, \forall \boldsymbol{x}(d \Rightarrow A) \rangle \xrightarrow{*} \langle X;\ c';\ \forall \boldsymbol{x}(d \Rightarrow A), !\Gamma' \rangle$.
– π ends with a promotion :

$$\frac{!\Gamma^\dagger \vdash_{\mathcal{C}} !c}{!\Gamma^\dagger \vdash_{\mathcal{C}} c}$$

By induction, $(\emptyset; \mathbf{1}; !\boldsymbol{\Gamma}) \xrightarrow{*} \langle \mathbf{X};\ \mathbf{d};\ !\boldsymbol{\Gamma}' \rangle$ with $\exists X.d^\dagger \vdash_{\mathcal{C}} c$ and $X \cap \mathcal{V}(c) = \emptyset$. To conclude notice that $!c \vdash_{\mathcal{C}} c$.
– π ends with a weakening : there are two subcases

$$\frac{\Gamma^\dagger \vdash_{\mathcal{C}} c}{\Gamma^\dagger, !d^\dagger \vdash_{\mathcal{C}} c} \qquad \text{or} \qquad \frac{\Gamma^\dagger \vdash_{\mathcal{C}} c}{\Gamma^\dagger, !\forall \boldsymbol{x}(d^\dagger \multimap A^\dagger) \vdash_{\mathcal{C}} c}$$

In the first subcase just notice that $\langle \emptyset;\ \mathbf{1};\ !d, \Gamma \rangle \xrightarrow{*} \langle \emptyset;\ \mathbf{1};\ \Gamma \rangle$. The second case is obvious since $!\forall \boldsymbol{x}(d^\dagger \multimap A^\dagger)$ is the translation of a persistent ask.
– π ends with a contraction : there are two subcases

$$\frac{\Gamma^\dagger, !d^\dagger, !d^\dagger \vdash_{\mathcal{C}} c}{\Gamma^\dagger, !d^\dagger \vdash_{\mathcal{C}} c} \qquad \text{or} \qquad \frac{\Gamma^\dagger, !\forall \boldsymbol{x}(d^\dagger \multimap A^\dagger), !\forall \boldsymbol{x}(d^\dagger \multimap A^\dagger) \vdash_{\mathcal{C}} c}{\Gamma^\dagger, !\forall \boldsymbol{x}(d^\dagger \multimap A^\dagger) \vdash_{\mathcal{C}} c}$$

In the first case notice that $!d \dashv\vdash !d \otimes !d$. The second one is trivial since any constraint consumed by twice the same persistent ask can be consumed by only one. □

B.2 Modules as Agents

Proposition 4. *Let Γ be a multiset of constraints, c a constraint and X a set of variables.*
(1) If $\Gamma \not\vdash_{\mathcal{C}} \mathbf{0}$ then $\mathcal{A}(X,\Gamma) \subset X \cup \mathcal{V}(\Gamma)$.
(2) If $\Gamma \vdash_{\mathcal{C}} c \otimes d$ then $\mathcal{A}(X,\Gamma) \supset \mathcal{A}(X,c)$.

Proof. (1) is proved by a very simple induction on the proof of $y \in \mathcal{A}(X,\Gamma)$. Just notice that we have supposed that if x and y are two syntactically different variables and $c \vdash_{\mathcal{C}} x = y \otimes d$ then $\{x,y\} \subset \mathcal{V}(X,\Gamma)$.

The result (2) is proved by a double induction first on the set $\mathcal{V}(c) \setminus X$ then on the proof π of $y \in \mathcal{A}(X,c)$:

- For the rule $=$ just notice that if $c \vdash_{\mathcal{C}} c = y \otimes \top$ then $\Gamma \vdash_{\mathcal{C}} c = y \otimes \top$.
- For the rule $\exists$ just suppose w.l.o.g. that $z \notin \mathcal{V}(\Gamma) \cup X$.
- For the rule ψ: we know thanks to (1) that if $z \notin X \cup \mathcal{V}(c)$ their is no proof of $z \notin \mathcal{A}(X,c)$. Hence we can suppose that $z \notin \mathcal{V}(c) \setminus X$. To conclude simply use the induction hypothesis.
- For Σ_P we have $c = p(x,\boldsymbol{s})$, $y \in \mathcal{V}(\boldsymbol{s})$ and $x \in X$. By a simple induction on the proof $\Gamma \vdash_{\mathcal{C}} c \otimes d$ we infer that either $\Gamma \vdash_{\mathcal{C}} \mathbf{0}$ or $p(x',\boldsymbol{s}')$ such that $\Gamma \vdash_{\mathcal{C}} (x',s') = (x,s) \otimes \top$. The first subcase is trivial since $\mathcal{A}(X,\mathbf{0}) = V$. The second subcase is immediate using rules Σ_P and ψ.
- The cases of the rules $\otimes - L$, $\otimes - R$ and ! are trivial.

Proof (lemma 1). The result is proved by induction on the proof π of $\Gamma \vdash_{\mathcal{C}} d[\boldsymbol{x}\backslash\boldsymbol{t}]$:

- π is a logical axiom : (w.l.o.g. d is atomic). There are two subcases:
  - d is in $\mathcal{D}$: just remark that $\boldsymbol{t} = \emptyset$.
  - d is in $\mathcal{P}$: d is of the form $p(y,\boldsymbol{s})$ with $y \neq x$ then we trivially have $\mathcal{V}(t) \subset \mathcal{A}(\mathcal{V}_P(d),\Gamma)$.
- π is a non-logical axiom. Once again there are two subcases:
  - $(\Gamma,d) \in \mathcal{D}^*$: again just remark that $\boldsymbol{t} = \emptyset$.
  - π is of the form $p(y,\boldsymbol{z}) \otimes (y,\boldsymbol{z}) = (y'\boldsymbol{z}') \vdash_{\mathcal{C}} p(y',\boldsymbol{z}')$ where all variables are pairwise disjoint. Clearly $y' \notin \boldsymbol{x}$ and then $\mathcal{V}(t) \subset \boldsymbol{z}'$. Here just notice that $\boldsymbol{z}' \subset \mathcal{A}(y',\Gamma)$.
- π ends with a cut:

$$\frac{\overline{c \vdash_{\mathcal{C}} e} \quad \Gamma, e \vdash d[\boldsymbol{x}\backslash\boldsymbol{t}]}{\Gamma, c \vdash_{\mathcal{C}} d[\boldsymbol{x}\backslash\boldsymbol{t}]} \qquad \frac{\Gamma \vdash_{\mathcal{C}} c \quad \overline{c \vdash d[\boldsymbol{x}\backslash\boldsymbol{t}]}}{\Gamma \vdash_{\mathcal{C}} d[\boldsymbol{x}\backslash\boldsymbol{t}]}$$

  Use previous proposition.
- π ends with the introduction of an ILL constant : Just notice that ILL constant $\mathbf{1}$, $\mathbf{0}$ and $\top$ are not modularized.
- π ends with a right introduction of $\otimes$:

$$\frac{\Gamma_1 \vdash_{\mathcal{C}} d_1[\boldsymbol{x}\backslash\boldsymbol{t}] \quad \Gamma_2 \vdash_{\mathcal{C}} d_2[\boldsymbol{x}\backslash\boldsymbol{t}]}{\Gamma_1, \Gamma_2 \vdash_{\mathcal{C}} d_1[\boldsymbol{x}\backslash\boldsymbol{t}] \otimes d_2[\boldsymbol{x}\backslash\boldsymbol{t}]}$$

For the left hand side premise, let $\{\boldsymbol{x}_1, \overline{\boldsymbol{x}_1}\}$ be a partition of $\boldsymbol{x}$ such as $\boldsymbol{x}_1 = \boldsymbol{x} \cap \mathcal{V}_P(d_1)$ and $\overline{\boldsymbol{x}_1} = \boldsymbol{x} \setminus \mathcal{V}_P(d_1)$ and $\{\boldsymbol{t}_1, \overline{\boldsymbol{t}_1}\}$ the corresponding partition of $\boldsymbol{t}$. Hence, we have $\mathcal{V}_P(d_1[\overline{\boldsymbol{x}_1} \backslash \overline{\boldsymbol{t}_1}]) = \mathcal{V}_P(d_1)$ and therefore by induction hypothesis $\mathcal{V}_P(\boldsymbol{t}_1) \subset \mathcal{A}(\mathcal{V}_P(d_1), \Gamma_1)$. Using the same reasoning we infer that $\mathcal{V}_P(\boldsymbol{t}_2) \subset \mathcal{A}(\mathcal{V}_P(d_2), \Gamma_2)$ where t_2 is the subsequence of $\boldsymbol{t}$ which corresponds to $\boldsymbol{x}_2 = \boldsymbol{x} \cap \mathcal{V}_P(d_2)$. By hypothesis we have $\boldsymbol{x} \cap \mathcal{V}_P(d_1 \otimes d_2)$ an then $\mathcal{V}_P(\boldsymbol{t}) \subset (\mathcal{A}(\mathcal{V}_P(d_1), \Gamma_1) \cup \mathcal{A}(\mathcal{V}_P(d_2), \Gamma_2)) \subset \mathcal{A}(\mathcal{V}_P(d_1 \otimes d_2), (\Gamma_1, \Gamma_2))$.

– π ends with a left introduction of $\exists$:

$$\frac{\Gamma \vdash_{\mathcal{C}} d[\boldsymbol{x} \backslash \boldsymbol{t}][z \backslash s]}{\Gamma \vdash_{\mathcal{C}} (\exists z.d)[\boldsymbol{x} \backslash \boldsymbol{t}]}$$

Without loss of generality we have $z \notin \boldsymbol{x}$ and $z \notin \mathcal{V}(\boldsymbol{t})$. There are two subcases:

- $\boldsymbol{z} \in \mathcal{V}_P(d)$: by induction hypothesis $\mathcal{V}(\boldsymbol{t}, s) \subset \mathcal{A}(\mathcal{V}_P(d), \Gamma)$. Therefore $\mathcal{V}(\boldsymbol{t}) \subset \mathcal{A}(\mathcal{V}_P(\exists z.d), \Gamma)$ since $z \notin \mathcal{V}(\boldsymbol{t})$.
- $\boldsymbol{z} \notin \mathcal{V}_P(d)$: by induction hypothesis, $\mathcal{V}(\boldsymbol{t}) \subset \mathcal{A}(\mathcal{V}_P(d), \Gamma)$. Because $z \notin \mathcal{V}(\boldsymbol{t})$ we have $\mathcal{V}(\boldsymbol{t}) \subset \mathcal{A}(\mathcal{V}_P(\exists z.d), \Gamma)$.

– π ends with a right introduction of $\exists$

$$\frac{\Gamma, c \vdash_{\mathcal{C}} d[\boldsymbol{x} \backslash \boldsymbol{t}]}{\Gamma(\exists z.c) \vdash_{\mathcal{C}} d[\boldsymbol{x} \backslash \boldsymbol{t}]}$$

By induction hypothesis, we have $\mathcal{V}(\boldsymbol{t}) \subset \mathcal{A}(\mathcal{V}_P(d), (\Gamma, c))$. Without loss of generality $z \notin \mathcal{V}(d)$, hence $\mathcal{V}(\boldsymbol{t}) \subset \mathcal{A}(\mathcal{V}_P(d), (\Gamma, \exists z.c))$. □

Concurrency of the Preflow-Push Algorithm in Constraint Handling Rules

Marc Meister

Fakultät für Ingenieurwissenschaften und Informatik
Universität Ulm, Germany

Abstract. Parallel implementations of the preflow-push algorithm are usually realised by low-level programming. A high-level and rule-based design offers declarative concurrency for speed-up by parallel execution. We present and analyse a concise implementation of the preflow-push algorithm by four rules in the Constraint Handling Rules language.

1 Introduction

Constraint Handling Rules (CHR) [6] is a concurrent, committed-choice, rule-based language which was originally created as a declarative logic constraint language. Its main features are guarded rules which transform multi-sets of constraints (atomic formulas) into simpler ones until they are solved. CHR programs enjoy declarative concurrency [12] that allows speed-up by parallel execution [7] similar to the logical parallelism in the *chemical reaction metaphor* [3]. Over the last decade, CHR has matured into a general-purpose, declarative programming language with many applications [15], e.g. the classic union-find algorithm has been implemented with optimal complexity in CHR [14].

The *preflow-push algorithm* [8, 5] solves the maximal flow problem. Applications for finding a maximal flow are manifold and found in, e.g. transportation planning and resource management. In constraint programming, the maximal flow is needed for the efficient handling of the global `alldifferent` and global cardinality constraints [11, 16].

Specification of the imperative preflow-push algorithm as a CHR program, under the objective to enjoy a high speed-up by parallel execution from its declarative concurrency, is a challenge.

Contributions and Overview. Our concise, concurrent CHR implementation of the preflow-push algorithm is optimised for speed-up by parallel and distributed execution.

- We recall the preflow-push algorithm and provide the necessary background for readers not familiar with CHR in Section 2.
- We present our concise, concurrent CHR implementation of the preflow-push algorithm and prove its correctness in Section 3.
- We use the declarative concurrency of CHR for parallel speed-up and provide experimental results in Section 4.
- We briefly relate our work in Section 5.

Supplementary Online Information. Our CHR implementation is available for online testing at `http://www.informatik.uni-ulm.de/pm/index.php?id=141`.

2 Preliminaries

Readers familiar with the preflow-push algorithm and CHR may skip this section.

2.1 Generic Preflow-Push Algorithm

A *flow network* is a complete, directed graph $G = (V, E)$ with two distinguished nodes source $\mathrm{s} \in V$ and sink $\mathrm{t} \in V$. In this paper, we restrict ourselves to capacities $c : E \to \{0, 1\}$ (these are needed for the bipartite matching underlying the implementation of the global `alldifferent` constraint) and require $c(u, v) + c(v, u) \leq 1$ for any two nodes $u, v \in V$. A *flow* in a flow network is a function $f : E \to \{-1, 0, 1\}$ obeying capacity restriction $\forall u, v \in V : f(u, v) \leq c(u, v)$, skew symmetry $\forall u, v \in V : f(u, v) = -f(v, u)$, and flow conservation $\forall u \in V \setminus \{\mathrm{s}, \mathrm{t}\} : \sum_{v \in V} f(u, v) = 0$. The *maximum-flow problem* is solved by any flow f that maximises the flow value $\sum_{u \in V} f(u, \mathrm{t})$ from source to sink.

The preflow-push algorithms [8] (see, e.g., [5] for a detailed introduction) employ a global label *height* $h : V \to \mathbf{N}$, use the *residual capacity* $r : E \to \{0, 1\}$ defined by $c(u, v) - f(u, v)$ for each edge (u, v), and apply the two actions *push* and *lift* in arbitrary order – hence "generic" preflow-push algorithm, cf. Figure 1. During the execution of the algorithm, the flow conservation property is relaxed: For a *pre-flow* f the *excess flow* $e(u) = \sum_{v \in V} f(v, u)$ can be positive for any node $u \in V$. When no action is applicable any more, the pre-flow f is a valid flow and solves the maximum-flow problem.

Initialise by $f \leftarrow 0$, except for $f(\mathrm{s}, u) \leftarrow c(\mathrm{s}, u)$, $h \leftarrow 0$, except for $h(\mathrm{s}) \leftarrow \#V - 2$, and $e \leftarrow 0$, except for $e(u) \leftarrow c(\mathrm{s}, u)$ for all nodes $u \in V$.

"Push along edge (u, v)" applies when $e(u) > 0$, $r(u, v) = 1$, and $h(u) > h(v)$. Then do push one unit of flow from u to v by updating $e(u) \leftarrow e(u) - 1$, $f(u, v) \leftarrow f(u, v) + 1$, $r(u, v) \leftarrow 0$, $e(v) \leftarrow e(v) + 1$, $f(v, u) \leftarrow f(v, u) - 1$, and $r(v, u) \leftarrow 1$.

"Lift node u" applies when $u \neq \mathrm{s}$, $u \neq \mathrm{t}$, $e(u) > 0$, and all residual edges are upward, i.e. $r(u, v) = 1$ and $h(u) \leq h(v)$.
Then do lift node u by updating $h(u) \leftarrow 1 + \min\{h(v) : r(u, v) = 1\}$.

Fig. 1. Generic preflow-push algorithm

Lift and *push* actions on disjoint parts of the graph are *concurrent* and allow *parallel execution*. However, the *lift* action requires a *sequential program* to compute the minimum height *before* updating.

2.2 Constraint Handling Rules

Constraint Handling Rules (CHR) [6, 15] is a concurrent, committed-choice, rule-based logic programming language. We distinguish between two different kinds

of constraints: *built-in constraints* which are solved by a given constraint solver, and *user-defined constraints* which are defined by the rules in a CHR program. This distinction allows one to embed and utilise existing constraint solvers as well as side-effect-free host language statements. As we trust the built-in black-box constraint solvers, there is no need to modify or inspect them.

A *CHR program* is a finite set of rules $R \mathbin{@} H_1 \setminus H_2 \Leftrightarrow G \mid B$. Each rule has a unique identifier R, the *head* $H_1 \setminus H_2$ is a non-empty multi-set conjunction of user-defined constraints, the *guard* G is a conjunction of built-in constraints, and the *body* B is a goal. A *goal* is a multi-set conjunction of built-in and user-defined constraints. We omit the trivial guard expression "*true* |". A rule R is a *simpagation rule* if both head expressions H_1 and H_2 are non-empty. If expression H_1 is empty, we have a *simplification rule* and write $R \mathbin{@} H_2 \Leftrightarrow G \mid B$. We do not use *propagation rules* with empty head expression H_2 in this paper.

The *operational semantics* of CHR is defined by a state transition system where states are multi-set conjunctions of atomic constraints. Any of the rules that are applicable can be applied and rule application cannot be undone since CHR is a committed-choice language. A rule $R \mathbin{@} H_1 \setminus H_2 \Leftrightarrow G \mid B$ is applicable in state $\langle H_1' \wedge H_2' \wedge C\rangle$ if the built-in constraints C_b of C imply that H_1' matches H_1, H_2' matches H_2, and the guard G is entailed under this matching, cf. (1). The consistent, predicate logic, built-in constraint theory CT contains Clark's syntactic equality and (for this work) basic arithmetic for integers.

$$\begin{array}{ll} \text{IF} & R \mathbin{@} H_1 \setminus H_2 \Leftrightarrow G \mid B \text{ with new variables } \bar{X} \\ \text{AND} & CT \models (\forall)\, C_b \rightarrow \exists \bar{X}\, (H_1 = H_1' \wedge H_2 = H_2' \wedge G) \\ \text{THEN} & \langle H_1' \wedge H_2' \wedge C\rangle \rightarrowtail_R \langle H_1' \wedge G \wedge B \wedge H_1 = H_1' \wedge H_2 = H_2' \wedge C\rangle \end{array} \tag{1}$$

If applied, a rule *replaces* the matched user-defined constraints of the head expression H_2 in the state by the body of the rule. Rules are applied until exhaustion, i.e. the CHR run-time system (which actually runs a CHR program P by selecting applicable rules $R \in P$ and matching constraints) computes the reflexive transitive closure $\rightarrowtail_P^*$ of $\rightarrowtail_P$. The derivation $\langle C\rangle \rightarrowtail_P^* \langle C'\rangle$ has initial goal C, answer C', and *derivation length* defined by the number of rule applications.

CHR rules have an immediate linear logic declarative semantics where the guard implies a logical equality between the l.h.s. and r.h.s. of a rule [4]. CHR programs enjoy natural declarative concurrency, cf. [12]. Each rule application is a logically independent calculation allowing sequential or parallel implementation.

3 Concurrent Preflow-Push in Constraint Handling Rules

We present our CHR version of the preflow-push and prove its correctness.

3.1 Preflow-Push in CHR: Four Rules

For each node u we keep its current *height* $h(u)$, its current excess flow $e(u)$, and its number $n(u)$ of outward capacity edges – defined by $\#\{(u,v) : c(u,v) = 1\}$ – in binary constraints $\mathtt{h}(u, h(u))$, $\mathtt{e}(u, e(u))$, and $\mathtt{n}(u, n(u))$.

By $r(u,v) = c(u,v) - f(u,v)$, it suffices to keep track of *residual edges* (u,v) with $r(u,v) = 1$ as for any two nodes u and v with $c(u,v) + c(v,u) = 1$ either (u,v) or (v,u) is a residual edge. A constraint $\mathtt{r}(u,v,k)$ encodes a *residual edge* (u,v) which is *unchecked* iff $k < h(u)$ and *checked* iff $k = h(u)$, i.e. the *sort* of the residual edge (u,v) depends on the height $h(u)$ of node u.

Each $\mathtt{m}(u,m,c)$ constraint contains a candidate value m of the auxiliary minimum computation, i.e. for any node u, there can be none, one, or several $\mathtt{m}(u,.,.)$ constraints in the same state.

We encode the initial pre-flow as a conjunction of the user-defined constraints $\mathtt{h}/2$, $\mathtt{e}/2$, $\mathtt{n}/2$, and $\mathtt{r}/3$. In the initial goal, residual edges (u,v) are *unchecked*. We then apply the rules of program P, cf. Figure 2, exhaustively and we will show that the answer of the CHR derivation $\rightarrowtail_P^*$ encodes a solution to the maximal-flow problem. Our upper-case rule variables are mnemonic, e.g. variable H_U is matched by $h(u)$ for a rule application. We frequently abbreviate "$\wedge$" to ",".

$$
\begin{aligned}
&\text{up} \;@\; \mathtt{h}(U,H_U), \mathtt{h}(V,H_V) \setminus \mathtt{r}(U,V,K) \\
&\qquad \Leftrightarrow H_U \leq H_V, K < H_U \mid \mathtt{m}(U,H_V,1), \mathtt{r}(U,V,H_U) \\
&\text{push} \;@\; \mathtt{h}(U,H_U), \mathtt{h}(V,H_V) \setminus \mathtt{e}(U,E_U), \mathtt{e}(V,E_V), \mathtt{r}(U,V,K) \\
&\qquad \Leftrightarrow 0 < E_U, H_V < H_U \mid \mathtt{e}(U,E_U-1), \mathtt{e}(V,E_V+1), \mathtt{m}(V,H_U,1), \mathtt{r}(V,U,H_V) \\
&\text{lift} \;@\; \mathtt{n}(U,N_U), \mathtt{e}(U,E_U) \setminus \mathtt{h}(U,H_U), \mathtt{m}(U,M,C) \\
&\qquad \Leftrightarrow U \neq \mathrm{s}, U \neq \mathrm{t}, 0 < E_U, C = N_U + E_U \mid \mathtt{h}(U,M+1) \\
&\text{min} \;@\; \mathtt{m}(U,M,C), \mathtt{m}(U,M',C') \Leftrightarrow \mathtt{m}(U,\min(M,M'),C+C')
\end{aligned}
$$

Fig. 2. Program P: Preflow-push in Constraint Handling Rules

Rule push realises the *push* action and all four CHR rules together realise a variant of the *lift* action of the generic preflow-push algorithm. We explain each rule of program P in turn.

Application of up recognises an *upward* edge (u,v) with $h(u) \leq h(v)$ by replacing the *unchecked* edge $\mathtt{r}(u,v,k) \wedge k < h(u)$ with the *checked* edge $\mathtt{r}(u,v,h(u))$ and inserts one $\mathtt{m}(u,h(v),1)$ constraint.

Application of push *pushes flow* along a *downward* edge (u,v) by replacing the *unchecked* edge $\mathtt{r}(u,v,k)$ with the *checked* edge $\mathtt{r}(v,u,h(v))$ in the reverse direction and inserts one $\mathtt{m}(v,h(u),1)$ constraint.

Application of lift *lifts* a node u to its new height $h(u) \leftarrow m+1$ by replacing $\mathtt{h}(u,h(u))$ with $\mathtt{h}(u,m+1)$ and removes the $\mathtt{m}(u,m,c)$ constraint.

Application of min keeps the *minimal* value of the two candidate values m and m' by replacing $\mathtt{m}(u,m,c) \wedge \mathtt{m}(u,m',c')$ with $\mathtt{m}(u,\min(m,m'),c+c')$.

Example 1. For the easy flow graph $V = \{\mathrm{s},x,\mathrm{t}\}$ with positive capacities $c(\mathrm{s},x) = c(x,\mathrm{t}) = 1$ we compute $\rightarrowtail_P^*$.

$$
\begin{aligned}
&\langle \mathtt{r}(x,\mathrm{s},-1), \mathtt{r}(x,\mathrm{t},-1), \mathtt{h}(\mathrm{s},1), \mathtt{h}(x,0), \mathtt{h}(\mathrm{t},0), \mathtt{e}(\mathrm{s},0), \mathtt{e}(x,1), \mathtt{e}(\mathrm{t},0), C' \rangle \\
\rightarrowtail_P^* &\langle \mathtt{r}(x,\mathrm{s},1), \mathtt{r}(\mathrm{t},x,0), \mathtt{h}(\mathrm{s},1), \mathtt{h}(x,1), \mathtt{h}(\mathrm{t},0), \mathtt{e}(\mathrm{s},0), \mathtt{e}(x,0), \mathtt{e}(\mathrm{t},1), C' \rangle
\end{aligned}
$$

The maximal flow (from source s via node x to sink t, with flow value 1) is given by tracking back residual edges starting from t. ($C' = \mathtt{n}(\mathrm{s},0) \wedge \mathtt{n}(x,1) \wedge \mathtt{n}(\mathrm{t},0)$)

3.2 Program P: Instance of the Generic Preflow-Push Algorithm

We show that program P (cf. Figure 2) is an instance of the the generic preflow-push algorithm (cf. Figure 1), i.e. we prove that the derivation $\rightarrow_P^*$ terminates with a solution to the maximal flow-problem. To this end, we prove that each state of the derivation encodes a valid pre-flow where application of rules push or lift actually changes the pre-flow while application of rules up or min is invariant to the pre-flow. Rules up and push exhaustively remove all *unchecked* residual edges for any overflowing node u by replacing them with *checked* edges, either from u to v or – when pushing flow downwards – from v to u. Application of rule lift changes all outward edges to *unchecked*.

Let (S_s) denote the sequence of computed states by program P, i.e. $S_0 \rightarrowtail_P S_1 \rightarrowtail_P \cdots$ starting from the initial pre-flow encoded in state S_0. We define the *number of outward residual edges* $o(u)$ of a node u in state S of (S_s) by $S = \langle \bigwedge_{i=1}^{o(u)} \mathtt{r}(u, v_i, k_i), C' \rangle$ where C' does not contain any $\mathtt{r}(u,.,.)$ constraint. We define the *number of outward and checked residual edges* $c(u)$ of a node u in state S of (S_s) by $S = \langle \bigwedge_{i=1}^{c(u)} \mathtt{r}(u, v_i, h(u)), \mathtt{h}(u, h(u)), C' \rangle$ where C' does not contain any $\mathtt{r}(u,.,h(u))$ constraint. Clearly we have $c(u) \leq o(u)$ for any state.

Table 1. Effects of rule application and interaction with minimum computation

rule application	effect on pre-flow	no. of checked edges	interaction with min
up on edge (u,v)		$c(u) \leftarrow c(u)+1$	insert $\mathtt{m}(u, h(v), 1)$
push on edge (u,v)	push flow downward	$c(v) \leftarrow c(v)+1$	insert $\mathtt{m}(v, h(u), 1)$
lift on node u	increase height	$c(u) \leftarrow 0$	remove $\mathtt{m}(u, m, c)$

The effects of application of rules up, push, or lift on the number of *checked* edges and their interaction with the auxiliary min-computation by inserting and removing $\mathtt{m}/3$ constraints are summarised in Table 3.2. Note that receiving flow from a neighbour node does not increase the number of *unchecked* edges as such an edge is already marked as *checked*. For each node u at most $o(u)$ many rule applications are possible until *all* edges are marked as *checked*. We now make our argument formal.

Property 1. For all states S of (S_s) the following *invariants* hold.

(*i*) For $S = \langle \mathtt{n}(u, n(u)), \mathtt{e}(u, e(u)), C' \rangle$ we have $o(u) = n(u) + e(u)$.
(*ii*) For $S = \langle \mathtt{r}(u, v, k), \mathtt{h}(u, h(u)), C' \rangle$ we have $k \leq h(u)$.
(*iii*) For $S = \langle \mathtt{r}(u, v, h(u)), \mathtt{h}(u, h(u)), \mathtt{h}(v, h(v)), C' \rangle$ we have $h(u) \leq h(v)$.
(*iv*) For $S = \langle \bigwedge_{i=1}^{j} \mathtt{m}(u, m_i, c_i), \mathtt{h}(u, h(u)), \bigwedge_{i=1}^{k} (\mathtt{r}(u, v_i, h(u)) \wedge \mathtt{h}(v_i, h(v_i))), C' \rangle$ and C' does not contain any $\mathtt{m}(u,.,.)$ constraint we have $c(u) = \sum_{i=1}^{j} c_i$ and $h(u) \leq \min\{m_1, \ldots, m_j\} \leq \min\{h(v_i) : 1 \leq i \leq k\}$.

Proof. Properties (*i-iv*) hold in state S_0 by our encoding of the initial pre-flow. For the induction step we consider the effects of $S_s \rightarrowtail_R S_{s+1}$ for each rule $R \in P$ under the induction hypothesis that properties (*i-iv*) hold in state S_s.

Application of up on edge (u, v). For $h(u) \leq h(v) \wedge k < h(u)$ we have

$$\begin{aligned} S_s =& \langle \mathtt{h}(u, h(u)), \mathtt{h}(v, h(v)), \mathtt{r}(u, v, k), C' \rangle \text{ and} \\ S_{s+1} =& \langle \mathtt{h}(u, h(u)), \mathtt{h}(v, h(v)), \mathtt{m}(u, h(v), 1), \mathtt{r}(u, v, h(u)), C' \rangle \ . \end{aligned}$$

Property (i) is not affected. By *insertion* of a $\mathtt{r}(u, v, h(u))$ constraint with $h(u) \leq h(v)$, properties $(ii\text{-}iii)$ also hold in state S_{s+1}. Note that replacing an *unchecked* edge constraint $\mathtt{r}(u, v, k) \wedge k < h(u)$ with the *checked* edge constraint $\mathtt{r}(u, v, h(u))$ updates $c(u)$ to $c(u)+1$. As we insert one $\mathtt{m}(u, h(v), 1)$ constraint with $h(u) \leq h(v)$, property (iv) also holds in state S_{s+1}.

Application of push on edge (u, v). For $h(v) < h(u)$ we have

$$\begin{aligned} S_s =& \langle \mathtt{h}(u, h(u)), \mathtt{h}(v, h(v)), \mathtt{e}(u, e(u)), \mathtt{e}(v, e(v)), \mathtt{r}(u, v, k), C' \rangle \text{ and} \\ S_{s+1} =& \langle \mathtt{h}(u, h(u)), \mathtt{h}(v, h(v)), \mathtt{e}(u, e(u) - 1), \mathtt{e}(v, e(v) + 1), \\ & \mathtt{m}(v, h(u), 1), \mathtt{r}(v, u, h(v)), C' \rangle \ . \end{aligned}$$

By replacing the residual edge (u, v) with the residual edge (v, u), updating $e(u) \leftarrow e(u) - 1$, and updating $e(v) \leftarrow e(v) + 1$, property (i) also holds in state S_{s+1}. By (iii), the edge constraint $\mathtt{r}(u, v, k)$ is *unchecked* in state S_s and properties $(ii\text{-}iv)$ hold in state S_{s+1} by similar argumentation as given for application of rule up on edge (v, u).

Application of lift on node u. For $c = n(u) + e(u)$ we have

$$\begin{aligned} S_s =& \langle \mathtt{n}(u, n(u)), \mathtt{e}(u, e(u)), \mathtt{h}(u, h(u)), \mathtt{m}(u, m, c), C' \rangle \text{ and} \\ S_{s+1} =& \langle \mathtt{n}(u, n(u)), \mathtt{e}(u, e(u)), \mathtt{h}(u, m + 1), C' \rangle \ . \end{aligned}$$

By property (iv), lifting of node u strictly increases its height. As only applications of rule lift affect height, we deduce that heights never decrease. By (i) and (iv), the guard $c = n(u) + e(u)$ ensures that *all* outward residual edges of node u are *checked* in state S_s and are hence *unchecked* in state S_{s+1} by (ii).

Application of rule lift has no effect on property (i). As heights never decrease, property (ii) also holds in state S_{s+1}. We consider two cases for property (iii): Note that there are no *checked and outward* residual edges of node u in state S_{s+1} and that for any *inward* residual edge (v, u) of node u, property $h(v) \leq h(u)$ also holds in S_{s+1} as heights never decrease. In both cases, property (iii) also holds in state S_{s+1}. By properties (i) and (iv), the guard $c = n(u) + e(u)$ ensures that $\mathtt{m}(u, c, m)$ is the only $\mathtt{m}(u, ., .,)$ constraint in state S_s. As state S_{s+1} neither contains any $\mathtt{m}(u, ., .,)$ nor any checked edge $\mathtt{r}(u, v, h(u))$ constraint, property (iv) also holds in state S_{s+1}.

Application of min. We have

$$\begin{aligned} S_s =& \langle \mathtt{m}(u, m, c), \mathtt{m}(u, m', c'), C' \rangle \text{ and} \\ S_{s+1} =& \langle \mathtt{m}(u, \min(m, m'), c + c'), C' \rangle \ . \end{aligned}$$

Properties (*i-iii*) are not affected. Property (*iv*) also holds in state S_{s+1} due to associativity and commutativity of addition and minimum computation.

By induction, properties (*i-iv*) are *invariants* for all states S of (S_s). □

Property 2. Rules push and lift implement the corresponding actions of the generic preflow-push algorithm.

Proof. For both rules, only overflowing nodes can be applicable due to the guards. The correspondence is immediate for rule push. The applicability of rule lift on node u with $u \neq \mathrm{s} \wedge u \neq \mathrm{t}$ depends on the inequality $c(u) \leq o(u) = n(u) + e(u)$. For $c(u) = o(u)$, invariants (i), (iii), and (iv) of Property 1 imply that all outward edges are *upward*. Due to (concurrent) lifting of neighbours, the computed minimum m for node u might be lower than the actual minimum of the neighbours' heights at the time of lifting, as we update to $m + 1$ with $h(u) \leq m \leq \min\{h(v) : r(u, v) = 1\}$ by invariant (iv) of Property 1. However, then the *lift* action is again applicable. □

Theorem 1. *Program P implements the generic preflow-push algorithm, i.e. the derivation $\rightarrow_P^*$ terminates with a solution to the maximal flow-problem.*

Proof. Based on the established Properties 1 and 2, rules push and lift update the pre-flow according to the general preflow-push algorithm. As the number of applications of rules up or min is bounded for each node u and height $h(u)$ – and the application of rule lift strictly increases a height label – the computation terminates. □

4 Concurrency for Parallel Speed-Up

We investigate parallel speed-up using the declarative concurrency of CHR and give some experimental results for our preflow-push implementation. We are interested in a theoretical parallel computation model which assumes an unbounded number of processors that communicate via a shared store.

4.1 Simulating Parallel Computations by Interleaving

The parallel CHR computation model is related to the *chemical reaction metaphor* [3] with molecules (constraints) interacting freely (in parallel) according to reaction rules (copies of the rules of our CHR program). We slightly extend its definition in [7] by combining *computations* which keep the overlapping constraints into one parallel computation.

Definition 1 (Parallel Rule Application in CHR). *Consider multisets of constraints A_1, A_2, B_1, B_2, H_1, H_2, and C, a CHR rule R, and multisets of CHR rules R_1 and R_2 (written $\|$-separated). A* parallel rule application $\rightrightarrows$ *is defined by the following two inference rules.*

$$\frac{\langle A_1\rangle \rightarrowtail_R \langle B_1\rangle}{\langle A_1\rangle \rightrightarrows_R \langle B_1\rangle} \qquad \frac{\langle H_1 \wedge C\rangle \rightrightarrows_{R_1} \langle B_1 \wedge C\rangle \quad \langle H_2 \wedge C\rangle \rightrightarrows_{R_2} \langle B_2 \wedge C\rangle}{\langle H_1 \wedge H_2 \wedge C\rangle \rightrightarrows_{R_1 \| R_2} \langle B_1 \wedge B_2 \wedge C\rangle} \quad (2)$$

Example 2. Consider r @ $\mathtt{a} \backslash \mathtt{a} \Leftrightarrow \mathit{true}$ which removes duplicate constraints. Then $\langle \mathtt{a}, \mathtt{a} \rangle \rightrightarrows_r \langle \mathtt{a} \rangle$, $\langle \mathtt{a}, \mathtt{a}, \mathtt{a} \rangle \rightrightarrows_{r \| r} \langle \mathtt{a} \rangle$, and $\langle \mathtt{a}, \mathtt{a}, \mathtt{a}, \mathtt{a}, \mathtt{a} \rangle \rightrightarrows_{r \| r \| r} \langle \mathtt{a}, \mathtt{a} \rangle$ are examples for parallel rule applications, while $\langle \mathtt{a}, \mathtt{a} \rangle \rightrightarrows_{r \| r} \langle \mathit{true} \rangle$ is *not* a parallel rule application. Note that for the simplification rule variant r' @ $\mathtt{a}, \mathtt{a} \Leftrightarrow \mathtt{a}$ of rule r, the transition $\langle \mathtt{a}, \mathtt{a}, \mathtt{a} \rangle \rightrightarrows_{r' \| r'} \langle \mathtt{a} \rangle$ is *not* a parallel rule application.

To facilitate the simulation of a parallel derivation $\langle C \rangle \rightrightarrows^* \langle C' \rangle$ as a sequential derivation $\langle C \rangle \rightarrowtail^* \langle C' \rangle$ by interleaving semantics [13, 12] we introduce *time-stamps* for sequential derivations. Note that interleaving is possible due to the monotonicity property of CHR [7, 1].

Definition 2 (Time-stamps). *We attach the time-stamp* 0 *to the initial goal constraints and adapt the state transition system: Rule* R *applies* at time t*, if the* maximum *of the time-stamps of* all matched head constraints *is* $t - 1$*. We then attach the time-stamp* t *to all newly inserted body constraints.* $[\ {}^{t}\!\rightarrowtail_R\ ]$

Example 3. Consider s @ $\mathtt{a} \backslash \mathtt{b} \Leftrightarrow \mathtt{a}$ and the goal $\mathtt{a} \wedge \mathtt{b} \wedge \mathtt{b}$. Selecting ${}^0\mathtt{a}$ for the second application of s yields $\langle\, {}^0\mathtt{a}\,,\, {}^0\mathtt{b}\,,\, {}^0\mathtt{b}\,\rangle\, {}^1\!\rightarrowtail \langle\, {}^0\mathtt{a}\,,\, {}^1\mathtt{a}\,,\, {}^0\mathtt{b}\,\rangle\, {}^1\!\rightarrowtail \langle\, {}^0\mathtt{a}\,,\, {}^1\mathtt{a}\,,\, {}^1\mathtt{a}\,\rangle$, and selecting ${}^1\mathtt{a}$ yields $\langle\, {}^0\mathtt{a}\,,\, {}^0\mathtt{b}\,,\, {}^0\mathtt{b}\,\rangle\, {}^1\!\rightarrowtail \langle\, {}^0\mathtt{a}\,,\, {}^1\mathtt{a}\,,\, {}^0\mathtt{b}\,\rangle\, {}^2\!\rightarrowtail \langle\, {}^0\mathtt{a}\,,\, {}^1\mathtt{a}\,,\, {}^2\mathtt{a}\,\rangle$.

Property 3. Any sequential derivation $\langle C \rangle \rightarrowtail^n \langle C' \rangle$ with rule application times $t_1, \ldots, t_n$ can be *combined* into a parallel derivation $\langle C \rangle \rightrightarrows^m \langle C' \rangle$ with *parallel derivation length* $m = \max\{t_1, \ldots, t_n\}$.

Proof. We combine all rule applications of $\rightarrowtail^*$ which occur *at the same time* t into *one parallel* rule application $\rightrightarrows$. As only constraints with earlier time-stamps are removed at time t no overlapping constraints are removed, cf. (2). □

Example 4. Derivations of Example 3 are combined to $\langle \mathtt{a}, \mathtt{b}, \mathtt{b} \rangle \rightrightarrows_{s \| s} \langle \mathtt{a}, \mathtt{a}, \mathtt{a} \rangle$ and $\langle \mathtt{a}, \mathtt{b}, \mathtt{b} \rangle \rightrightarrows_s \langle \mathtt{a}, \mathtt{a}, \mathtt{b} \rangle \rightrightarrows_s \langle \mathtt{a}, \mathtt{a}, \mathtt{a} \rangle$ according to Property 3.

We use a simple *heuristics* to achieve a low parallel derivation length. We *greedily* apply rules at *minimal time*, i.e. we *exhaustively apply rules for a given time* t before progressing to time $t + 1$.

4.2 Experimental Results

We tested program P for different inputs: A *random level graph* (x, y, c) is a rectangular grid of nodes with x rows and y columns where nodes have c outgoing capacity edges to randomly chosen nodes in the following row. The external source connects to each node in the top row and each node in the bottom row connects to the external sink.

We exhaustively apply rules of program P according to our *greedy heuristics*, prefer application of rule lift over application of rule push, and select constraints randomly. We define *speed-up* as sequential derivation length divided by parallel length. Note that *all* auxiliary computations (recognising upward edges and computation of minima) are included in the parallel length, cf. Table 4.2.

Table 2. Actions of the generic preflow-push, derivation lengths, and speed-up

level graphs	no. of lifts	no. of pushes	sequential length	parallel length	speed-up
$(10, 1, 1)$	10	10	51	30	1.7
$(5, 5, 3)$	26.9	31.8	293.4	39.0	8.1
$(3, 20, 10)$	63.0	64.0	1780.8	34.8	51.4

The linear chain $(10, 1, 1)$ from source to sink via 10 intermediate nodes has only few potential for parallelisation as *one* unit of flow is sequentially pushed, yet only 30 parallel rule applications suffice. As the more dense, random level 5×5, square grids have more edges per node, the concurrent and distributed recognition of upward edges and minimum computation pays off. The high total amount of 20 flow units contribute contribute to the high parallel speed-up for the even move dense, random level $(3, 20, 10)$, wide grids. Note that we achieve an average of 3.7 application of the generic *push* or *lift* actions per parallel rule application for them.

Summarising, speed-up depends on the total amount of flow units, its distribution on disjoint nodes, and the density of the flow network.

5 Related Work

In earlier work we presented a preliminary, hand-crafted, parallel version of the preflow-push in CHR where user-defined constraints are used extensively for the control, e.g. for means of concurrent locking by trailing (sets of) dependency graphs [10]. The total of 26 rules – including *propagation rules* and nitty-gritty details of the CHR compiler – are contrary to the high-level aims of easy understanding, reasoning, and optimisation.

Initially we tried an approach for the preflow-push that was successful for the classic union-find algorithm [7]. Using confluence analysis and program completion in order to achieve a *confluent* variant, however, was unsuccessful due to the number of required rules and the inherently *non-confluent* specification of the generic preflow-push algorithm.

Existing parallel implementations of the preflow-push [2, 9] are concerned about a *real-time* speed-up, are tailored to existing hardware by low-level imperative programming, and use sophisticated data-structures. This ad-hoc parallelism is contrary to our concise, declarative, and high-level design which requires only the multiset data-structure. A *real-time* parallel speed-up by our approach requires a parallel CHR version which is, unfortunately, not available.

6 Conclusion

We carefully designed the preflow-push algorithm for CHR with the intention to exploit its declarative concurrency for a high degree of parallel and distributed execution. Our design restricts the necessarily *sequential part* of the algorithm to a *single* CHR rule application. Our concise specification of the preflow-push

as an (executable) CHR program runs without sophisticated data- or control-structures. We showed its potential for significant parallel speed-up by experimental results.

Future Work. Allowing arbitrary and non-integral capacities requires to adapt our concise encoding of the residual graph. As *push* actions can then be *non-saturating*, i.e. there is less overflow than available residual capacity, both the preflow *and* the capacity graph are needed.

We plan to integrate both the gap heuristic and the periodic global relabelling heuristic [2] in order to include bigger test cases [9].

Our research is driven by the long term goal of a concurrent implementation of Régin's global `alldifferent` constraint [11, 16] in CHR which uses maximal matching (a special instance of the maximal-flow problem).

Acknowledgements. I thank Thom Frühwirth for many helpful comments.

References

1. S. Abdennadher, T. Frühwirth, and H. Meuss. Confluence and semantics of constraint simplification rules. *Constraints*, 4(2):133–165, 1999.
2. R. J. Anderson and J. C. Setubal. On the parallel implementation of Goldberg's maximum flow algorithm. In PAAA 1992, pages 168–177. ACM Press, 1992.
3. J.-P. Banâtre, P. Fradet, and Y. Radenac. Principles of chemical programming. In *RULE 2004*, volume 124 of *ENTCS*, pages 133–147. Elsevier, 2005.
4. H. Betz and T. Frühwirth. A linear-logic semantics for Constraint Handling Rules. In *CP 2005*, volume 3709 of *LNCS*, pages 137–151. Springer, 2005.
5. T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. The MIT Press, 2nd edition, 2001.
6. T. Frühwirth. Theory and Practice of Constraint Handling Rules. *J. Logic Programming*, 37(1–3):95–138, 1998.
7. T. Frühwirth. Parallelizing union-find in Constraint Handling Rules using confluence. In *ICLP 2005*, volume 3668 of *LNCS*, pages 113–127. Springer, 2005.
8. A. V. Goldberg and R. E. Tarjan. A new approach to the maximum-flow problem. *J. ACM*, 35(4):921–940, 1988.
9. D. S. Johnson and C. C. McGeoch, editors. *Network Flows and Matching: First DIMACS Implementation Challenge*, volume 12 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. AMS, 1993.
10. M. Meister. Fine-grained parallel implementation of the preflow-push algorithm in CHR. In *WLP 2006*, volume 1843-06-02 of *INFSYS Research Report*, pages 172–181. Technische Universität Wien, Austria, 2006.
11. J.-C. Régin. A filtering algorithm for constraints of difference in CSPs. In *AAAI 1994*, volume 1, pages 362–367. AAAI Press, 1994.
12. V. A. Saraswat. *Concurrent Constraint Programming*. The MIT Press, 1993.
13. D. A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. Allyn and Bacon, Inc., Boston (Mass.), USA, 1986.
14. T. Schrijvers and T. Frühwirth. Optimal union-find in Constraint Handling Rules. *J. Theory and Practice of Logic Programming*, 6(1&2):213–224, 2006.
15. T. Schrijvers and others. The Constraint Handling Rules (CHR) web page, 2007. `http://www.cs.kuleuven.ac.be/~dtai/projects/CHR/`.
16. W. J. van Hoeve. The `alldifferent` constraint: A survey. In *ERCIM Working Group on Constraints, 6th Annual Workshop, 2001, Prague, Czech Republic*, 2001.

Soft Constraint Problems with Incompleteness

Mirco Gelain, Maria Silvia Pini, Francesca Rossi, and Kristen Brent Venable

Dipartimento di Matematica Pura ed Applicata, Università di Padova, Italy
E-mail: {mgelain,mpini,frossi,kvenable}@math.unipd.it

Abstract. We consider soft constraint problems where some of the preferences may be unspecified. This models, for example, situations with several agents providing the data, or with possible privacy issues. In this context, we study how to find an optimal solution without having to wait for all the preferences. In particular, we define an algorithm to find a solution which is necessarily optimal, that is, optimal no matter what the missing data will be, with the aim to ask the user to reveal as few preferences as possible. Experimental results show that in many cases a necessarily optimal solution can be found by eliciting a small number of preferences.

1 Introduction

Traditionally, tasks such as scheduling, planning, and resource allocation have been tackled using several techniques, among which constraint reasoning is one of the winning ones: the task is represented by a set of variables, their domains, and a set of constraints, and a solution of the problem is an assignment to all the variables in their domains such that all constraints are satisfied. Preferences or objective functions have been used to extend such scenario and allow for the modelling of constraint optimization, rather than satisfaction, problems. However, what is common to all these approaches is that the data (variables, domains, constraints) are completely known before the solving process starts.

On the contrary, the increasing use of web services and in general of multi-agent applications demands for the formalization and handling of data that is only partially known when the solving process works, and that can be added later, for example via elicitation. In many web applications, data may come from different sources, which may provide their piece of information at different times. Also, in multi-agent settings, data provided by some agents may be voluntarily hidden due to privacy reasons, and only released if needed to find a solution to the problem.

Recently, some lines of work have addressed these issues by allowing for open settings in CSPs: both open CSPs [7, 9] and interactive CSPs [13] work with domains that can be partially specified, and in dynamic CSPs [6] variables, domains, and constraints may change over time. It has been shown that these approaches are closely related. In fact, interactive CSPs can be seen as a special case of both dynamic and open CSPs [12].

Here we consider the same issues but we focus on constraint optimization problems rather than CSPs, thus looking for an optimal solution rather than any solution. In

particular, we consider problems where constraints are replaced by soft constraints, in which each assignment to the variables of the constraint has an associated preference coming from a preference set [1]. In this setting, for the purpose of this paper we assume that variables, domains, and constraint topology are given at the beginning, while the preferences can be partially specified and possibly added during the solving process.

Constraint optimization has also been considered in the context of open CSPs in a cost-minimization setting [8]. However, the incompleteness considered is on which values belong to domains, and not on the preference of tuples.

There are several application domains where such setting is useful. One regards the fact that quantitative preferences, and needed in soft constraints, may be difficult and tedius to provide for a user. Another one concerns multi-agent settings, where agents agree on the structures of the problem by they may provide their preferences on different parts of the problem at different times. Finally, some preferences can be initially hidden because of privacy reasons.

Formally, we take the soft constraint formalism when preferences are totally ordered and we allow for some preferences to be left unspecified. Although some of the preferences can be missing, it could still be feasible to find an optimal solution. If not, then we ask the user and we start again from the new problem with some added preferences.

More precisely, we consider two notions of optimal solution: *possibly optimal* solutions are assignments to all the variables that are optimal in *at least one way* in which currently unspecified preferences can be revealed, while *necessarily optimal* solutions are assignments to all the variables that are optimal in *all ways* in which currently unspecified preferences can be revealed. This notation comes from multi-agent preference aggregation [14], where, in the context of voting theory, some preferences are missing but still one would like to declare a winner.

Given an incomplete soft constraint problem (ISCSP), its set of possibly optimal solutions is never empty, while the set of necessarily optimal solutions can be empty. Of course what we would like to find is a necessarily optimal solution, to be on the safe side: such solutions are optimal regardless of how the missing preferences would be specified. However, since such a set may be empty, in this case there are two choices: either to be satisfied with a possibly optimal solution, or to ask users to provide some of the missing preferences and try to find, if any, a necessarily optimal solution of the new ISCSP. In this paper we follow this second approach, and we repeat the process until the current ISCSP has at least one necessarily optimal solution.

To achieve this, we employ an approach based on branch and bound which first checks whether the given problem has a necessarily optimal solution (by just solving the completion of the problem where all unspecified preferences are replaced by the worst preference). If not, then finds the most promising among the possibly optimal solutions (where the promise is measured by its preference level), and asks the user to reveal the missing preferences related to such a solution. This second step is then repeated until the current problem has a necessarily optimal solution.

We implemented our algorithm and we tested it against classes of randomly generated binary fuzzy ISCSPs, where, beside the usual parameters (number of variables, domain size, density, and tightness) we added a new parameters measuring the percentage of unspecified preferences in each constraint and domain. The experimental results

show that in many cases a necessarily optimal solution can be found by eliciting a small amount of preferences. In particular, in no case the percentage of elicited preferences exceeded 35% of the total number of missing preferences.

2 Soft constraints

A soft constraint [1] is just a classical constraint [5] where each instantiation of its variables has an associated value from a (totally or partially ordered) set. This set has two operations, which makes it similar to a semiring, and is called a c-semiring. More precisely, a c-semiring is a tuple $\langle A, +, \times, \mathbf{0}, \mathbf{1}\rangle$ such that: A is a set, called the carrier of the c-semiring, and $\mathbf{0}, \mathbf{1} \in A$; $+$ is commutative, associative, idempotent, $\mathbf{0}$ is its unit element, and $\mathbf{1}$ is its absorbing element; $\times$ is associative, commutative, distributes over $+$, $\mathbf{1}$ is its unit element and $\mathbf{0}$ is its absorbing element.

Consider the relation $\leq_S$ over A such that $a \leq_S b$ iff $a+b = b$. Then: $\leq_S$ is a partial order; $+$ and $\times$ are monotone on $\leq_S$; $\mathbf{0}$ is its minimum and $\mathbf{1}$ its maximum; $\langle A, \leq_S\rangle$ is a lattice and, for all $a, b \in A$, $a + b = lub(a, b)$. Moreover, if $\times$ is idempotent, then $\langle A, \leq_S\rangle$ is a distributive lattice and $\times$ is its glb.

Informally, the relation $\leq_S$ gives us a way to compare (some of the) tuples of values and constraints. In fact, when we have $a \leq_S b$, we will say that *b is better than a*. Thus, $\mathbf{0}$ is the worst value and $\mathbf{1}$ is the best one.

Given a c-semiring $S = \langle A, +, \times, \mathbf{0}, \mathbf{1}\rangle$, a finite set D (the domain of the variables), and an ordered set of variables V, a constraint is a pair $\langle def, con\rangle$ where $con \subseteq V$ and $def : D^{|con|} \rightarrow A$. Therefore, a constraint specifies a set of variables (the ones in con), and assigns to each tuple of values of D of these variables an element of the semiring set A. A soft constraint satisfaction problem (SCSP) is just a set of soft constraints over a set of variables.

Many known classes of satisfaction or oprimization problem can be cast in this formalism. A classical CSP is just an SCSP where the chosen c-semiring is: $S_{CSP} = \langle\{false, true\}, \vee, \wedge, false, true\rangle$. On the other hand, fuzzy CSPs [15, 11] can be modeled in the SCSP framework by choosing the c-semiring: $S_{FCSP} = \langle|0, 1|, max, min, 0, 1\rangle$. For weighted CSPs, the semiring is $S_{WCSP} = \langle\Re^+, min, +, +\infty, 0\rangle$. Here preferences are interpreted as costs from 0 to $+\infty$, which are combined with the sum and compared with min. Thus the optimization criterion is to minimize the sum of costs. For probabilistic CSPs [10], the semiring is $S_{PCSP} = \langle[0, 1], max, \times, 0, 1\rangle$. Here preferences are interpreted as probabilities ranging from 0 to 1, which are combined using the product and compared using max. Thus the aim is to maximize the joint probability.

Given an assignment t to all the variables of an SCSP, we can compute its preference value $pref(t)$ by combining the preferences associated by each constraint to the subtuples of the assignments referring to the variables of the constraint. More precisely, $pref(P, s) = \Pi_{\langle idef,con\rangle \in C} def(s_{\downarrow con})$, where Π refers to the $\times$ operation of the semiring and $s_{\downarrow con}$ is the projection of tuple s on the variables in con.

For example, in fuzzy CSPs, the preference of a complete assignment is the minimum preference given by the constraints. In weighted constraints, it is instead the sum of the costs given by the constraints.

An optimal solution of an SCSP is then a complete assignment t such that there is no other complete assignment t'' with $pref(t) <_S pref(t'')$. The set of optimal solutions of an SCSP P will be written as $Opt(P)$.

3 Incomplete Soft Constraint Problems (ISCSPs)

Informally, an incomplete SCSP, written ISCSP, is an SCSP where the preferences of some tuples in the constraints, and/or of some values in the domains, are not specified. In detail, given a set of variables V with finite domain D, and c-semiring $S = \langle A, +, \times, 0, 1\rangle$ with a totally ordered carrier, we extend the SCSP framework to incompleteness by the following definitions.

Definition 1 (incomplete soft constraint). *Given a set of variables V with finite domain D, and a c-semiring $\langle A, +, \times, 0, 1\rangle$, an incomplete soft constraint is a pair $\langle idef, con\rangle$ where $con \subseteq V$ is the scope of the constraint and $idef : D^{|con|} \longrightarrow A \cup \{?\}$ is the preference function of the constraint. All tuples mapped into $?$ by $idef$ are called incomplete tuples.*

In an incomplete soft constraint, the preference function can either specify the preference value of a tuple by assigning a specific element from the carrier of the c-semiring, or leave such preference unspecified. Formally, in the latter case the associated value is $?$. A soft constraint is a special case of an incomplete soft constraint where all the tuples have a specified preference.

Example 1. Assume a travel agency is planning Alice and Bob's honeymoon, having only some information on their preferences. Alice and Bob live in the US, the candidate destinations are the Maldive islands and the Caribbean, and they can decide to go by ship or by plane. Going to Maldives by ship would take very long, thus they have a high preference to go by plane and a low preference to go by ship. Also, they have been told that a cruise in the Caribbean is very nice. Thus they have a high preference to go there by ship. However, they don't have any preference on going there by plane. We can model this scenario by using the fuzzy c-semiring $\langle [0, 1], max, min, 0, 1\rangle$, two variables T (standing for $Transport$) and D (standing for $Destination$) with domains $D(T) = \{p, sh\}$ (p stands for $plane$ and sh for $ship$) and $D(D) = \{m, c\}$ (m stands for $Maldives$, c for $Caribbean$), and an incomplete soft constraint $\langle idef, con\rangle$ with $con = \{T, D\}$ and with preference function as shown in Figure 1. The only incomplete tuple is (p, c).

Definition 2 (incomplete soft constraint problem (ISCSP)). *An incomplete soft constraint problem is a pair $\langle C, V, D\rangle$ where C is a set of incomplete soft constraints over the variables in V with domain D. Given an ISCSP P, we will denote with $IT(P)$ the set of all incomplete tuples in P.*

Definition 3 (completion). *Given an ISCSP P, a completion of P is an SCSP P' obtained from P by associating to each incomplete tuple in every constraint an element of the carrier of the c-semiring. A completion is partial if some preference remains unspecified. We will denote with $C(P)$ the set of all possible completions of P and with $PC(P)$ the set of all its partial completions.*

Example 2. Consider again Example 1. Assume that for the considered season the Maldives are slightly preferrable to the Caribbean. Also, Alice and Bob don't mind travelling by plane and have never travelled by ship. Thus they have a reasonably high

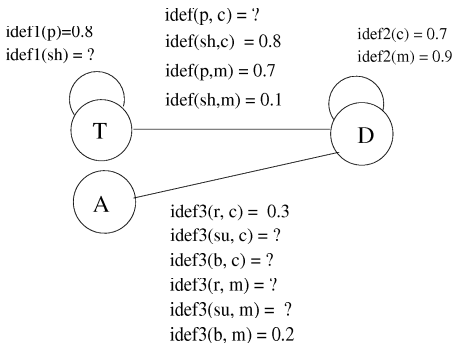


Fig. 1. An ISCSP.

preference to plane as a way of transport, while they don't give any preference to ship. Moreover, as far as accommodations, which can be in a standard room, or a suite, or a bungalow, assume that a suite in the Maldives is too expensive for the young couple while a standard room in the Caribbean is not special enough for a honeymoon. To model this new information we use a variable A (standing for $Accommodation$) with domain $D(A) = \{r, su, b\}$ (r stands for $room$, su for $suite$ and b for $bungalow$), and three constraints: two unary incomplete soft constraints, $\langle idef1, \{T\}\rangle$, $\langle idef2, \{D\}\rangle$ and a binary incomplete soft constraint $\langle idef3, \{A, D\}\rangle$. Their definition is shown in Figure 1. The set of incomplete tuples of this problem is $IT(P) = \{(sh), (p, c), (su, c), (su, m), (r, m), (b, c)\}$.

Definition 4 (preference of an assignment). *Given an ISCSP $P = \langle C, V, D\rangle$ and an assignment s to all its variables we denote with $pref(P, s)$ the preference of s in P. In detail, $pref(P, s) = \Pi_{<idef,con>\in C|idef(s_{\downarrow con})\neq ?} idef(s_{\downarrow con})$.*

The preference of an assignment s in an incomplete problem is thus obtained by combining the known preferences associated to the projections of the assignment, that is, of the appropriated subtuples in the constraints. The projections which have unspecified preferences are simply ignored. Given an assignment s to all the varibales, the set of its projections with unspecified preference is denoted by $it(s)$.

Example 3. Consider the two assignments $s_1 = (p, m, b)$ and $s_2 = (p, m, su)$, we have that $pref(P, s_1)) = min(0.8, 0.7, 0.9, 0.2) = 0.2$, while $pref(P, s_2) = min($ $0.8, 0.7, 0.9) = 0.7$. However, while the preference of s_1 is fixed, since none of its projections is incomplete, the preference of s_2 may become lower that 0.7 depending on the preference of the incomplete tuple (su, m).

As shown by the example, the presence of incompleteness generates a partition of the set of assignments into two sets: those which have a certain preference which is independent of how incompleteness is resolved, and those whose preference is only an upperbound, in the sense that it can be lowered in some completions.

Given an ISCSP P, we will denote the first set of assignments as $Fixed(P)$ and the second with $Unfixed(P)$. In Example 3, $Fixed(P) = \{s_1\}$, while all other assignments belong to $Unfixed(P)$.

In SCSPs we have that an assignment is an optimal solution if its global preference is undominated. This notion can be generalized to the incomplete setting. In particular, when some preferences are unknown, we will speak of necessarily and possibly optimal solutions, that is, assignments which are undominated in all (resp., some) completions.

Definition 5 (necessarily and possibly optimal solution). *Given an ISCSP* $P = \langle C, V, D\rangle$*, an assignment* $s \in D^{|V|}$ *is a necessarily (resp, possibly) optimal solution iff* $\forall Q \in C(P)$ *(resp.,* $\exists Q \in C(P)$ *such that)* $\forall s' \in D^{|V|}$, $pref(Q, s') \not> pref(Q, s)$.

Given an ISCSP P, we will denote with $NOS(P)$ (resp., $POS(P)$) the set of necessarily (resp., possibly) optimal solutions of P. Notice that, while $POS(P)$ is never empty, in general $NOS(P)$ may be empty. In particular, $NOS(P)$ is empty whenever the available preferences do not allow to determine the relation between an assignment and all the others.

Example 4. In the ISCSP P of Figure 1, we can easily see that $NOS(P) = \emptyset$ since, given any assignment, it is possible to construct a completion of P in which it is not an optimal solution. On the other hand, $POS(P)$ contains all assignments not including tuple (sh, m). In fact, such a tuple has preference 0.1 and it drowns the preference of any assignment containing it below the preference of fixed solution $s_1 = (p, m, b)$ (i.e., below 0.2). Thus, in all completions, s_1 dominates any such assignment. Instead, for any assignment s not including (sh, m), we can complete P in order to make s optimal, for example by setting the preferences of all the incomplete tuples of s to $pref(s, P)$ and the preferences of all other incomplete tuples to 0.

4 Characterizing POS(P) and NOS(P)

In this section we investigate how to characterize the set of necessarily and possibly optimal solutions of an ISCSP given the preferences of the optimal solutions of two of the completions of P. In particular, given an ISCSP P defined on c-semiring $\langle A, +, \times, \mathbf{0}, \mathbf{1}\rangle$, we consider:

- the SCSP $P_0 \in C(P)$, called the $\mathbf{0}$-completion of P, obtained from P by associating preference $\mathbf{0}$ to each tuple of $IT(P)$.
- the SCSP $P_1 \in C(P)$, called the $\mathbf{1}$-completion of P, obtained from P by associating preference $\mathbf{1}$ to each tuple of $IT(P)$.

Let us indicate respectively with $pref_0$ and $pref_1$ the preference of an optimal solution of P_0 and P_1. Due to the monotonicity of $\times$, and since $\mathbf{0} \leq \mathbf{1}$, we have that $pref_0 \leq pref_1$.

In the following theorem we will show that, if $pref_0 > \mathbf{0}$, there is a necessarily optimal solution of P iff $pref_0 = pref_1$, and in this case $NOS(P)$ coincides with the set of optimal solutions of P_0.

Theorem 1. *Given an ISCSP* P *and the two completions* $P_0, P_1 \in C(P)$ *as defined above, if* $pref_0 > \mathbf{0}$ *we have that:*

- $NOS(P) \neq \emptyset$ *iff* $pref_1 = pref_0$*;*

– *if* $NOS(P) \neq \emptyset$ *then* $NOS(P) = Opt(P_0)$.

Proof. Since we know that $pref_0 \leq pref_1$, if $pref_0 \neq pref_1$ then $pref_1 > pref_0$. We prove that, if $pref_1 > pref_0$, then $NOS(P) = \emptyset$. Let us consider any assignment s of P. Due to the monotonicity of $\times$, for all $P' \in C(P)$, we have $pref(P', s) \leq pref(P_1, s) \leq pref_1$.

– If $pref(P_1, s) < pref_1$, then s is not in $NOS(P)$ since P_1 is a completion of P where s is not optimal.
– If instead $pref(P_1, s) = pref_1$, then, since $pref_1 > pref_0$, we have $s \in Unfixed($ $P)$. Thus we can consider completion P_1' obtained from P_1 by associating preference $\mathbf{0}$ to the incomplete tuples of s. In P_1' the preference of s is $\mathbf{0}$ and the preference of an optimal solution of P_1' is, due to the monotonicity of $\times$, at least that of s in P_0, that is $pref_0 > \mathbf{0}$. Thus $s \notin NOS(P)$.

Next we consider when $pref_0 = pref_1$. Clearly $NOS(P) \subseteq Opt(P_0)$, since any assignment which is not optimal in P_0 is not in $NOS(P)$. We will show that $NOS(P) \neq \emptyset$ by showing that any $s \in Opt(P_0)$ is in $NOS(P)$. Let us assume, on the contrary, that there is $s \in Opt(P_0)$ such that $s \notin NOS(P)$. Thus there is a completion P' of P with an assignment s' with $pref(P', s') > pref(P', s)$. By construction of P_0, any assignment $s \in Opt(P_0)$ must be in $Fixed(P)$. In fact, if it had some incomplete tuple, its preference in P_0 would be $\mathbf{0}$, since $\mathbf{0}$ is the absorbing element of $\times$. Since $s \in Fixed(P)$, $pref(P', s) = pref(P_0, s) = pref_0$. By construction of P_1 and monotonicity of $\times$, we have $pref(P_1, s') \geq pref(P', s')$. Thus the contradiction $pref_1 \geq pref(P_1, s') \geq pref(P', s') > pref(P', s) = pref_0$. This allows us to conclude that $s \in NOS(P) = Opt(P_0)$. □

In the theorem above we have assumed that $pref_0 > \mathbf{0}$. The case in which $pref_0 = \mathbf{0}$ needs to be treated separetly. We consider it in the following theorem.

Theorem 2. *Given ISCSP* $P = \langle C, V, D \rangle$ *and the two completions* $P_0, P_1 \in C(P)$ *as defined above, assume* $pref_0 = \mathbf{0}$. *Then:*

– *if* $pref_1 = \mathbf{0}$, $NOS(P) = D^{|V|}$;
– *if* $pref_1 > \mathbf{0}$, $NOS(P) = \{s \in Opt(P_1) | \forall s' \in D^{|V|}$ *with* $pref(P_1, s') > \mathbf{0}$ *we have* $it(s) \subseteq it(s')\}$.

The formal proof is omitted for lack of space. However, we give the informal intuition. In words, the theorem above says that, if $pref_0 = \mathbf{0}$ and $pref_1 > \mathbf{0}$, then an assignment is a necessarily optimal solution only if it is optimal in P_1 and if the set of its incomplete tuples is contained in the set of incomplete tuples of all other assignments in $Unfixed(P)$. Intuitively, if some assignment s' has an incomplete tuple which is not part of another assignment s, then we can make s' dominate s in a completion by setting all the incomplete tuples of s' to $\mathbf{1}$ and all the remaining incomplete tuples of s to $\mathbf{0}$. In such a completion s is not optimal. Thus s is not a necessarily optimal solution.

However, if the tuples of s are a subset of the incomplete tuples of all other assignments then it is not possible to lower s without lowering all other tuples even further. This means that s is a necessarily optimal solution.

We now turn our attention to possible optimal solutions. Given a c-semiring $\langle A, +, \times, \mathbf{0}, \mathbf{1}\rangle$, it has been shown in [2] that either the $\times$ operator is idempotent or it is strictly monotonic. In the following two theorems we show that such a distinction on the c-semiring plays a key role in the characterization of $POS(P)$ where P is an ISCSP.

In particular, if $\times$ is idempotent, then the possible optimal solutions are the assignments with preference in P between $pref_0$ and $pref_1$. If, instead, $\times$ is strictly monotonic, then the possibliy optimal solutions have preference in P between $pref_0$ and $pref_1$ and dominate all the assignments which have as set of incomplete tuples a subset of their incomplete tuples.

Theorem 3. *Given an ISCSP P defined on a c-semiring with idempotent $\times$ and the two completions $P_0, P_1 \in C(P)$ as defined above, if $pref_0 > \mathbf{0}$ we have that: $POS(P) = \{s \in D^{|V|} | pref_0 \leq pref(P,s) \leq pref_1\}$.*

Proof. First we show that any s such that $pref_0 \leq pref(P,s) \leq pref_1$ is in $POS(P)$. Let us consider the completion of P, P', obtained by associating preference $pref(P,s)$ to all the incomplete tuples of s and $\mathbf{0}$ to all other incomplete tuples of P. For any other assignment s' we can show that it never dominates s:

- $s' \in Fixed(P)$ and thus $pref(P', s') = pref(P_0, s') \leq pref_0 \leq pref(P,s)$;
- $s' \in Unfixed(P)$ and
  - $it(s') \not\subseteq it(s)$, then $pref(P', s') = \mathbf{0}$ since in P' the incomplete tuples in $it(s')$ which are not in $it(s)$ have been associated to preference $\mathbf{0}$;
  - $it(s') \subseteq it(s)$. By construction of P' and since $\times$ is idempotent and associative we have that: $pref(P', s) = (pref(P,s) \times (\Pi_{|it(s)|} pref(P,s))) = pref(P,s)$ and $pref(P', s') = (pref(P,s') \times (\Pi_{|it(s')|} pref(P,s))) = pref(P, s') \times pref(P,s)$. Since $\times$ is intensive, $pref(P', s') = (pref(P,s') \times pref(P, s)) \leq pref(P,s) = pref(P', s)$.

Thus in P' no assignment dominates s. This means that $s \in POS(P)$.

We will now show that if $s \; in POS(P)$, $pref_0 \leq pref(P,s) \leq pref_1$. By construction of P_1 and due to monotonicity of $\times$ we have that there is no assignment s with $pref(P,s) > pref_1$. Thus, to conclude the proof of the theorem, we only need to show that any assignment s such that $pref(P,s) < pref_0$ is not in $POS(P)$. First let us notice that $\forall P' \in C(P)$ and for any assignment s we have that $pref(P', s) \leq pref(P,s)$ due to the intensive property of $\times$. Let us now consider any assignment s' such that $pref(P_0, s') = pref_0$. By construction of P_0, any assignment $s' \in Opt(P_0)$ must be in $Fixed(P)$, otherwise if it had some incomplete tuple its preference in P_0 would be $\mathbf{0}$, since $\mathbf{0}$ is the absorbing element of $\times$. Since $s' \in Fixed(P)$, $pref(P', s') = pref(P_0, s') = pref_0$, $\forall P' \in C(P)$. Thus, $\forall P' \in C(P)$, $pref(P', s) \leq pref(P,s) < pref_0 = pref(P', s')$. This shows that if $pref(P,s) < pref_0$ then $s \notin POS(P)$. □

Theorem 4. *Given an ISCSP P defined on a c-semiring with a strictly monotonic $\times$ and the two completions $P_0, P_1 \in C(P)$ as defined above, if $pref_0 > \mathbf{0}$ we have that: an assignment $s \in POS(P)$ iff $pref_0 \leq pref(P,s) \leq pref_1$ and $pref(P,s) = max\{\, pref(P,s') |\, it(s') \subseteq it(s)\}$.*

Proof. Let us first show that if assignment s is such that $pref_0 \leq pref(P,s) \leq pref_1$ and $pref(P,s) = max\{pref(P,s')|it(s') \subseteq it(s)\}$ it is in $POS(P)$. We must show there is a completion of P where s is undominated. Let us consider completion P' obtained associating preference $\mathbf{1}$ to all the tuples in $it(s)$ and $\mathbf{0}$ to all the tuples in $IT(P) \backslash it(s)$. First we notice that $pref(P',s) = pref(P,s)$, since $\mathbf{1}$ is the unit element of $\times$. Let us consider any other assignment s'. Then we have one of the following:

- $it(s') = \emptyset$, which means that $s' \in Fixed(P)$ and thus $pref(P',s') = pref(P_0,s') \leq pref_0 \leq pref(P,s) = pref(P',s)$;
- $it(s') \not\subseteq it(s)$, which means that there is at least one incomplete tuple of $it(s')$ which is associated to $\mathbf{0}$. Since $\mathbf{0}$ is the absorbing element of $\times$, $pref(P',s') = \mathbf{0}$ and thus $pref(P',s') < pref_0 \leq pref(P',s)$;
- $it(s') \subseteq it(s)$, in this case $pref(P',s') = pref(P,s')$ since all tuples in $it(s')$ are associated to $\mathbf{1}$ in P'. However since $pref(P,s) = max\{pref(P,s')|it(s') \subseteq it(s)\}$, $pref(P',s') \leq pref(P',s)$.

We can thus conclude that s is not dominated by any assignment in P'. Hence $s \in POS(P)$.

Let us now prove the other direction by contradiction. If $pref(P,s) < pref_0$ then we can conclude as in the previous proof. We must prove that if $pref_0 \leq pref(P,s) \leq pref_1$ and $pref(P,s) < max\{pref(P,s')|it(s') \subseteq it(s)\}$ then s is not in $POS(P)$. In any completion P' of P we have that $pref(P',s) = pref(P,s) \times it\text{-}pref(P',s)$ and $pref(P',s') = pref(P,s') \times it\text{-}pref(P',s')$ where $it\text{-}pref(P',s)$ (resp. $it\text{-}pref(P',s')$) is the combination of the preferences associated to the incomplete tuples in $it(s)$ (resp. $it(s')$). Since $it(s') \subseteq it(s)$, for any completion P' we have that $it\text{-}pref(P',s) \leq it\text{-}pref(P',s')$. Moreover, let s'' be such that $pref(P,s'') = max\{pref(P,s')|it(s') \subseteq it(s)\}$. Then we have that for any completion P', $pref(P',s'') > pref(P',s)$ since $pref(P,s'') > pref(P,s)$ and $it\text{-}pref(P',s'') \geq it\text{-}pref(P',s)$ and $\times$ is strictly monotonic. Thus, if $pref_0 \leq pref(P,s) \leq pref_1$ and $pref(P,s) < max\{pref(P,s')|it(s') \subseteq it(s)\}$, then s is not in $POS(P)$. □

In constrast to $NOS(P)$, when $pref_0 = \mathbf{0}$ we can immediately conclude that $POS(P) = D^{|V|}$, independently of the nature of $\times$, since all assignments are optimal in P_0.

Corollary 1. *Given an ISCSP $P = \langle C, V, D\rangle$, if $pref_0 = \mathbf{0}$, then $POS(P) = D^{|V|}$.*

The results given in this section can be summarized as follows:

- when $pref_0 = \mathbf{0}$
  - not enough information to compute $NOS(P)$ (by Theorem 2);
  - $POS(P) = D^{|V|}$ (by Corollary 1);
- when $pref_0 = pref_1 = \mathbf{0}$
  - $NOS(P) = D^{|V|}$ (by Theorem 2);
  - $POS(P) = D^{|V|}$ (by Corollary 1) ;
- when $\mathbf{0} = pref_0 < pref_1$
  - $NOS(P) = \{s \in Opt(P_1)|\forall s' \in D^{|V|}$ with $pref(P_1,s') > \mathbf{0}$ we have $it(s) \subseteq it(s')\}$ (by Theorem 2);

- $POS(P) = D^{|V|}$ (by Corollary 1);

– when $\mathbf{0} < pref_0 = pref_1$
- $NOS(P) = Opt(P_0)$ (by Theorem 1);
- if $\times$ is idempotent: $POS(P) = \{s \in D^{|V|} | pref_0 \leq pref(P, s) \leq pref_1\}$ (by Theorem 3);
- if $\times$ is strictly monotonic: $POS(P) = \{s \in D^{|V|} | pref_0 \leq pref(P, s) \leq pref_1, pref(P, s) = max\{\ pref(P, s') | it(s') \subseteq it(s)\}\}$ (by Theorem 4);

– when $\mathbf{0} < pref_0 < pref_1$
- $NOS(P) = \emptyset$ (by Theorem 1);
- $POS(P)$ as for the case when $\mathbf{0} < pref_0 = pref_1$.

5 A solver for ISCSPs

We want to find a necessarily optimal solution of the given problem, if it exists. In most cases, however, the available information will only allow to determine the set of possibly optimal solutions. In such cases, preference elicitation is needed to discriminate among such assignments in order to determine a necessarily optimal one of the new problem with the elicited preferences. In this section we describe an algorithm, called *Find-NOS*, to achieve this task.

Algorithm 1: Find-NOS

Input: an ISCSP P
Output: an ISCSP Q, an assignment s, a preference p
$P_0 \leftarrow P[?/0]$
$s_0, pref_0 \leftarrow BB(P_0, -)$
$s_1 \leftarrow s_0$
$pref_1 \leftarrow pref_0$
$s_{max} \leftarrow s_0$
$pref_{max} \leftarrow pref_0$
repeat
  $P_1 \leftarrow P[?/1]$
  if $pref_1 > pref_{max}$ **then**
    $s_{max} \leftarrow s_1$
    $pref_{max} \leftarrow pref_1$
  $s_1, pref_1 \leftarrow BB(P_1, pref_{max})$
  if $s_1 \neq nil$ **then**
    $S \leftarrow it(s_1)$
    $P \leftarrow$ Elicit(P, S)
    $pref_1 \leftarrow pref(P, s_1)$
until $s_1 \neq nil$;
return $P, s_{max}, pref_{max}$

Algorithm *Find-NOS* takes in input an ISCSP P over a totally ordered c-semiring and returns an ISCSP Q which is a partial completion of P, and an assignment $s \in$

$NOS(Q)$ together with its preference p. Given an ISCSP P, *Find-NOS* first checks if $NOS(P)$ is not empty, and, if so, it returns P, $s \in NOS(P)$, and its preference. If instead $NOS(P) = \emptyset$, it starts eliciting the preferences of some incomplete tuples.

In detail, *Find-NOS* first computes the $\mathbf{0}$-completion of P, written as $P[?/0]$, called P_0, and applies Branch and Bound (BB) to it. This allows to find an optimal solution of P_0, say s_0, and its preference $pref_0$.

In our notation, the application of the BB procedure has two parameters: the problem to which it is applied, and the starting bound. When BB is applied without a starting bound, we will write $BB(P, -)$. When the BB has finished, it returns a solution and its preference. If no solution is found, we assume that the returned items are both nil.

Variables s_1 and $pref_1$ (resp., s_{max} and $pref_{max}$) represent the optimal solution and the corresponding preference of the $\mathbf{1}$-completion of the current problem (written $P[?/1]$) (resp., the best solution and the corresponding preference found so far). At the beginning, such variables are initialized to s_0 and $pref_0$.

The main loop of the algorithm, achieved through the **repeat** command, computes the $\mathbf{1}$-completion, denoted by P_1, of the current problem. In the first iteration the condition of the first **if** is not satisfied since $pref_1 = pref_{max} = pref_0$. The execution thus proceeds by applying BB to P_1 with bound $pref_{max} = pref_0 \geq \mathbf{0}$. This allows us to find an optimal solution of P_1 and its corresponding preference, assigned to s_1 and $pref_1$. If BB fails to find a solution, s_1 is nil. Thus the second **if** is not executed and the algorithm exits the loop and returns P, $s_{max} = s_0$, and $pref_{max} = pref_0$.

If instead BB applied to P_1 with bound $pref_{max}$ does not fail, then we have that $pref_0 < pref_1$. Now the algorithm elicits the preference of some incomplete tuples, via procedure $Elicit$. This procedure takes an ISCSP and a set of tuples of variable assignments, and asks the user to provide the preference for such tuples, returning the updated ISCSP. The algorithm calls procedure $Elicit$ over the current problem P and the set of incomplete tuples of s_1 in P. After elicitation, the new preference of s_1 is computed and assigned to $pref_1$.

Since $s_1 \neq nil$, a new iteration begins, and BB is applied with initial bound given by the best preference between $pref_1$ and $pref_{max}$. Moreover, if $pref_1 > pref_{max}$, then s_{max} and $pref_{max}$ are updated to always contain the best solution and its preference. Iteration continues until the elicited preferences are enough to make BB fail to find a solution with a better preference w.r.t. the previous application of BB. At that point, the algorithm returns the current problem and the best solution found so far, together with its preference.

Theorem 5. *Given an ISCSP P in input, algorithm* Find-NOS *always terminates and returns an ISCSP Q such that $Q \in PC(P)$, an assignment $s \in NOS(Q)$, and its preference in Q.*

Proof. At each iteration, either $pref_{max}$ increases or, if it stays the same, a new solution will be found since after elicitation the preference of s_1 has not increased. Thus, either $pref_{max}$ is so high that BB doesn't find any solution, or all the optimal solutions have been considered. In both cases the algorithm exits the loop.

At the end of its execution, the algorithm returns the current partial completion of given problem and a solution s_{max} with the best preference seen so far $pref_{max}$. The

repeat command is exited when $s_1 = nil$, that is, when $BB(P[?/1], pref_{max})$ fails. In this situation, $pref_{max}$ is the preference of an optimal solution of the $\mathbf{0}$-completion of the current problem P. Since BB fails on $P[?/1]$ with such a bound, by monotonicity of the $\times$ operator, $pref_{max}$ is also the preference of an optimal solution of $P[?/1]$. By Theorems 1 and 2, we can conclude that $NOS(P)$ is not empty. If $pref_{max} = \mathbf{0}$, then $NOS(P)$ contains all the assignments and thus also s_0. The algorithm correctly returns the same ISCSP given in input, assignment s_0 and its preference $pref_0 = \mathbf{0}$. If instead $\mathbf{0} < pref_{max}$, again the algorithm is correct, since by Theorem 1 we know that $NOS(P) = Opt(P[?/0])$, and since $s_{max} \in Opt(P[?/0])$. □

Notice also that the algorithm performs preference elicitation only on solutions which are possibly optimal in the current partial completion of the given problem (and thus also in the given problem). In fact, by Theorems 3 and 4, any optimal solution of the $\mathbf{1}$-completion of the current partial completion Q is a possibly optimal solution of Q. Thus no useless work is done to elicit preferences related to solutions which cannot be necessarily optimal for any partial completion of the given problem. This also means that our algorithm works independently of the properties of the $\times$ operator.

6 Experimental setting and results

We have implemented Algorithm *Find-NOS* in Java and we have tested it on randomly generated ISCSPs with binary constraints and based on the Fuzzy c-semiring. To generate such problems, we use the following parameters:

- n: number of variables;
- m: cardinality of the domain of each variable;
- d: density of the constraints, that is, the percentage of binary constraints present in the problem w.r.t. the total number of possible binary constraints that can be defined on n variables;
- t: tightness, that is, the percentage of tuples with preference 0 in each constraint, w.r.t. the total number of tuples (m^2 since we have only binary constraints), and in each domain;
- i: incompleteness, that is, the percentage of incomplete tuples (formally, tuples with preference $?$) in each constraint and in each domain.

For example, if the generator is given in input $n = 10$, $m = 5$, $d = 50$, $t = 10$, and $i = 30$, it will generate a binary ISCSP with 10 variables, each with 5 elements in the domain, 22 constraints on a total of $45 = n(n-1)/2$, 2 tuples with preference 0 and 7 incomplete tuples over a total of 25 in each constraint, and 1 missing preference in each domain.

We have generated classes of ISCSPs by varying one parameter at a time, and fixing the other ones. The varying parameters are the number of variables, the density, and the incompleteness. When the number of variables varies (from $n = 5$ to $n = 20$, with step 3), we set $m = 5$, $d = 50$, $t = 10$, and $i = 30$. When we vary the density (from $d = 10$ to $d = 80$ with step 5), we set $n = 10$, $m = 5$, $t = 10$, and $i = 30$. Finally, when we vary the incompleteness (from $i = 10$ to $i = 80$ with step 5), we set $n = 10$, $m = 5$, $d = 50$, and $t = 10$.

In all the experiments, we have measured the number of tuples elicited by Algorithm *Find-NOS*. We also show the percentage of elicited tuples over the total number of incomplete tuples of the problem in input. For each fixed value of all the parameters, we show the average of the results obtained for 50 different problem instances, each given in input to *Find-NOS* 10 times. This setting is necessary since we have two kinds of randomness: the usual one in the generation phase and a specific one when eliciting preferences.

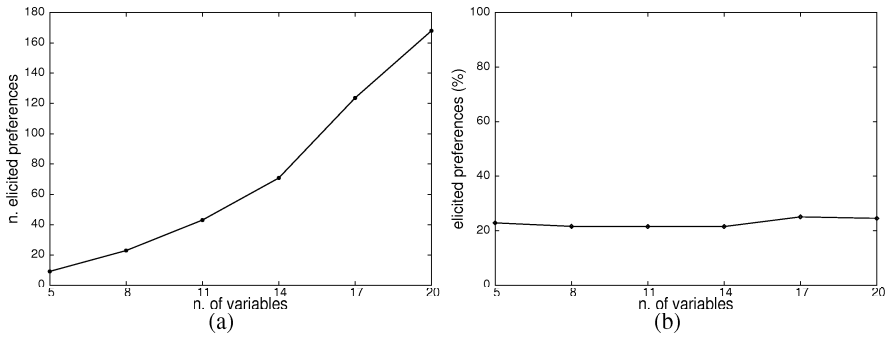


Fig. 2. Number and percentage of elicited preferences, as a function of the number of variables. Fixed parameters: $m = 5$, $d = 50$, $t = 10$, $i = 30$.

Figure 2 shows the absolute number and the percentage of elicited preferences when the number of variables varies. As expected, when the number of variables increases, the absolute number of elicited preferences increases as well, since there is a growth of the total number of incomplete tuples. However, if we consider the percentage of elicited tuples, we see that it is not affected by the increase in the number of variables. In particular, the percentage of elicited preferences remains stable around 22%, meaning that, regardless of the number of variables, the agent is asked to reveal only 22 preferences over 100 incomplete tuples. A necessarily optimal solution can be thus found leaving 88% of the missing preferences unrevealed.

Similar results are obtained when density varies (see Figure 3). We can see that the absolute number of elicited preferences grows when density increases. The maximum number of elicited preferences reached is however lower that the maximum reached when varying the variables (see Figure 2(a)). The reason for this is that the largest problems considered when varying the number of variables have more incomplete tuples than the largest obtained when varying the density. In fact, a problem with $n = 20$, given the fixed parameters, has around 685 incomplete tuples, 165 of which (about 22%) are elicited. On the other hand, a problem with $d = 80$, given the fixed parameters, has around 262 incomplete tuples, 55 (about 22%) of which are elicited. This is coherent with the fact that the results on the percentage of elicited preferences when varying the density and the number of variables are very similar.

The last set of experiments vary the percentage of incompleteness (see Figure 4). As for density and number of variables, the absolute number of elicited preferences grows when the percentage of incompleteness increases. The maximum number of elicited

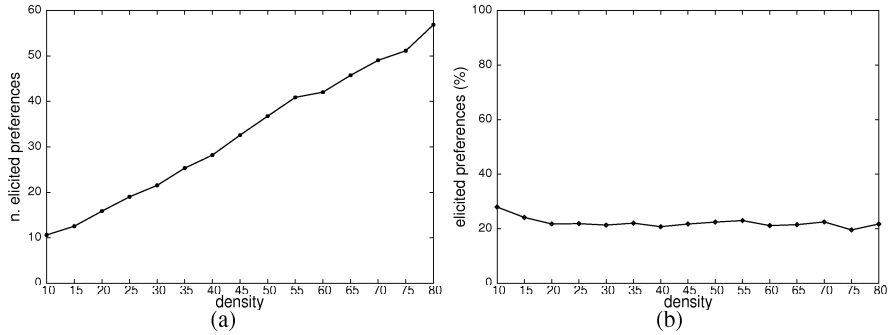


Fig. 3. Number and percentage of elicited preferences, as a function of the density. Fixed parameters: $n = 10$, $m = 5$, $t = 10$, $i = 30$.

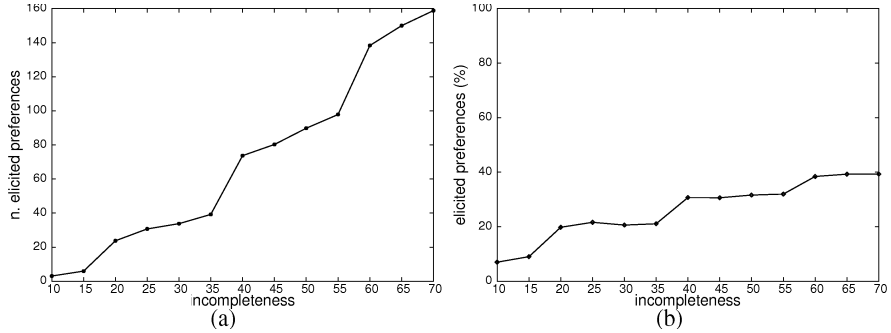


Fig. 4. Number and percentage of elicited preferences, as a function of the incompleteness. Fixed parameters: $n = 10$, $m = 5$, $t = 10$, $i = 30$.

preferences reached is close to that reached when varying the variables. However, the number of incomplete tuples of the problems with $i = 70$ is around 460 and thus smaller than that of problems with $n = 20$. Thus the percentage of elicited preferences is larger in problems with $i = 70$. This is confirmed by the corresponding result for the percentage of elicited preferences, which is shown to be around 35%. Additionally, the percentage of elicited preferences follows a slightly increasing trend as the percentage of incompleteness in the problem grows. However, it maintains itself below 35%, which means that in the worst case, where 70% of the tuples are incomplete, we are able to find a necessary optimal solution leaving 46% of the total number of tuples unspecified.

These experimental results show that it is indeed possible to find a necessarily optimal solution while forcing the user to reveal only a small percentage of the missing preferences. This is very promising both in terms of elicitation-related costs and also when concerned with privacy issues.

7 Conclusions and future work

We consider problems modelled via soft constraints with totally ordered and possibly unspecified preferences, and propose to solve them via an approach based on systematic search. Experimental results show that a small amount of preferences has to be revelead before being able to find an optimal solution.

Future work should consider partially ordered preferences and also other ways to express preferences, such as qualitative ones a la CP nets [4, 3], as well as other kinds of missing data, such as those considered in dynamic, interactive, and open CSPs. Moreover, other solving approaches can be considered, such as those based on local search rather than systematic search.

References

1. S. Bistarelli, U. Montanari, and F. Rossi. Semiring-based constraint solving and optimization. *Journal of the ACM*, 44(2):201–236, mar 1997.
2. S. Bistarelli, U. Montanari, F. Rossi, T. Schiex, G. Verfaillie, and H. Fargier. Semiring-based csps and valued csps: Frameworks, properties, and comparison. *Constraints*, 4(3), 1999.
3. C. Boutilier, R. I. Brafman, C. Domshlak, H. H. Hoos, and D. Poole. Cp-nets: A tool for representing and reasoning with conditional ceteris paribus preference statements. *J. Artif. Intell. Res. (JAIR)*, 21:135–191, 2004.
4. C. Boutilier, R. I. Brafman, H. H. Hoos, and D. Poole. Reasoning with conditional ceteris paribus preference statements. In *UAI*, pages 71–80, 1999.
5. R. Dechter. *Constraint processing*. Morgan Kaufmann, 2003.
6. R. Dechter and A. Dechter. Belief maintenance in dynamic constraint networks. In *AAAI*, pages 37–42, 1988.
7. B. Faltings and S. Macho-Gonzalez. Open constraint satisfaction. In *CP*, volume 2470 of *Lecture Notes in Computer Science*, pages 356–370. Springer, 2002.
8. B. Faltings and S. Macho-Gonzalez. Open constraint optimization. In *CP*, volume 2833 of *Lecture Notes in Computer Science*, pages 303–317. Springer, 2003.
9. B. Faltings and S. Macho-Gonzalez. Open constraint programming. *Artif. Intell.*, 161(1-2):181–208, 2005.
10. H. Fargier and J. Lang. Uncertainty in constraint satisfaction problems: a probalistic approach. In M. Clarke, R. Kruse, and S. Moral, editors, *Symbolic and Quantitative Approaches to Reasoning and Uncertainty, European Conference, ECSQARU'93, Granada, Spain, November 8-10, Proceedings*, volume 747 of *Lecture Notes in Computer Science*, pages 97–104. Springer, 1993.
11. H. Fargier, T. Schiex, and G. Verfaille. Valued Constraint Satisfaction Problems: Hard and Easy Problems. In *IJCAI-95*, pages 631–637. Morgan Kaufmann, 1995.
12. S. Macho González, C. Ansótegui, and P. Meseguer. On the relation among open, interactive and dynamic csp. In *The Fifth Workshop on Modelling and Solving Problems with Constraints (IJCAI'05)*, 2005.
13. E. Lamma, P. Mello, M. Milano, R. Cucchiara, M. Gavanelli, and M. Piccardi. Constraint propagation and value acquisition: Why we should do it interactively. In *IJCAI*, pages 468–477, 1999.
14. J. Lang, M. S. Pini, F. Rossi, K. B. Venable, and T. Walsh. Winner determination in sequential majority voting. In *IJCAI*, pages 1372–1377, 2007.
15. Z. Ruttkay. Fuzzy constraint satisfaction. In *Proceedings 1st IEEE Conference on Evolutionary Computing*, pages 542–547, Orlando, 1994.

Fuzzy Conditional Temporal Problems

M. Falda, F. Rossi, and K. B. Venable

Dept. of Pure and Applied Mathematics
University of Padova, ITALY

Abstract. Conditional Temporal Problems (CTPs) allow for the representation of temporal and conditional plans, dealing simultaneously with uncertainty and temporal constraints. In this paper, CTPs are generalized to CTPPs by adding preferences to the temporal constraints and by allowing fuzzy thresholds for the occurrence of some events. The usual consistency notions (strong, weak and dynamic) are then extended to encompass the new setting, and their corresponding testing algorithms are provided. We show that the complexity of the algorithms does not increase w.r.t. their classical counterparts for CTPs. We also show that our framework generalizes STPPUs as well, another temporal framework with uncertainty and preferences. This means that controllability in STPPUs can be translated to consistency in CTPPs, indicating a strong theoretical connection among the two formalisms.

1 Introduction

Many systems and applications need to be able to reason with alternative situations, plans, contexts and to know what holds in each of them. Moreover, they may have to set temporal constraints on events and actions. Conditional Temporal Problems (CTPs) [13] are a formalism that allows for modeling conditional and temporal plans which deal with the uncertainty arising from the outcome of observations and with complex temporal constraints. In CTPs the usual notion of consistency is replaced by three notions, weak, strong and dynamic consistency, which differ on the assumptions made on the knowledge available.

Another class of temporal reasoning problems that deals with similar scenarios are Simple Temporal Problems with Uncertainty (STPUs) [14]. In such problems the uncertainty lies in the lack of control the agent has over the time at which some events occur. Such events are said to be controlled by "Nature". In STPUs consistency is called controllability and, similarly to CTPs, there are three notions, weak, strong and dynamic controllability, based on different assumptions made on the uncontrollable variables. Despite the fact that consistency in CTPs and controllability in STPUs appear similar, their relation has not been formally investigated.

Furthermore, in rich application domains it is often necessary to handle not only temporal constraints and conditions, but also preferences over the execution of actions. Preferences have been added to STPUs in [10]; in addition to expressing uncertainty, in STPPUs contingent constraints can be soft, meaning that different preference levels are associated to different durations of events.

In this paper we introduce the CTPP model, an extension of CTPs which adds preferences to the temporal constraints and generalizes the simple Boolean conditions to

fuzzy rules; these rules activate the occurrence of some events on the basis of fuzzy thresholds. Moreover, also the activation of the events is characterized by a preference function over the domain of the event. This provides an additional gain in expressiveness, allowing one to model the dynamic aspect of preferences that change over time.

Quantitative temporal constraint problems have been used for many applications in practice, ranging from space applications (MAPGEN [1]) to temporal databases [2] and personal assistance (Autominder, [9]). We expect CTPPs to be useful in all of the above.

After defining CTPs with fuzzy preferences, we extend all the consistency notions of CTPs. Moreover, we provide algorithms for testing such new notions which are in the same complexity class as their classical counterparts. Finally, we show how the STPPUs are related to CTPPs by providing a mapping from STPPUs to CTPPs (and thus also from STPUs to CTPs) which preserves the controllability/consistency notions. In particular, such a mapping proves that CTPPs are a more expressive model. All proofs hae been omitted for lack of space.

2 Background

STPs and STPPs. A Simple Temporal Problem (STP) [4] is defined as a set of variables V, each of which corresponds to an instantaneous event, and a set E of constraints between the variables. The constraints are binary and are of the form $l_{ij} \leq x_i - x_j \leq u_{ij}$ with $x_i, x_j \in V$ and $l_{ij}, u_{ij} \in \Re$; l_{ij} and u_{ij} are called the bounds of the constraint.

Preferences have been introduced in STPs by [6], defining Simple Temporal Problems with Preferences (STPPs). In particular, a soft temporal constraint $< I, f >$ is specified by means of a preference function on the interval, $f : I \rightarrow [0, 1]$, where $I = [l_{ij}, u_{ij}]$. An STPP is said to be consistent with preference degree α if there exists an assignment of its variables that satisfies all constraints and that has preference α. The preference of an assignment is obtained by taking the $minimum$ of the preferences given by each constraint to the projection of the assignment onto its variables. An optimal solution is one such that there is no other solution with higher preference. Such a solution ca be found in polynomial time [6].

STPUs and STPPUs. STPUs [14] are STPs in which the temporal constraints are divided in two classes: those representing durations under the control of the agent (called requirement constraints) and those representing durations decided by "Nature" (called contingent constraints). Such a partition induces a similar partition over the variables. In [10] STPUs are extended to preferences by replacing STP constraints with soft temporal constraints. Thus an STPPU is a tuple $< N_e, N_c, L_r, L_c >$ where N_e is the set of executable timepoints, N_c is the set of contingent timepoints, L_r is a set of soft requirement constraints, and L_c is a set of soft contingent constraints. The notions of controllability of STPUs are extended to handle preferences. Here we focus on two of such notions. An STPPU is said to be α-strongly controllable is there is a fixed way to assign the values to the variables in N_e such that whatever Nature will choose for the variables in N_c the resulting assignment is either optimal (if Nature's choice prevents from achieving preference level α) or it has preference α. Optimal weak controllability simply requires the existence of an optimal way to assign values to the variables in N_e given an any assignment to those in N_c.

CTPs. CTPs [13] extend temporal constraint satisfaction problems [4] by adding observation variables and by conditioning the occurrence of some events on the presence of some properties of the environment. A CTP is a tuple $< V, E, L, OV, O, \mathcal{P} >$ where $\mathcal{P}$ is a set of Boolean atomic propositions, V is a set of variables, E is a set of temporal constraints between pairs of variables in V, $L : V \rightarrow \mathcal{Q}^*$ is a function attaching conjunctions of literals in $\mathcal{Q} = \{p_i : p_i \in P\} \cup \{\neg p_i : p_i \in P\}$ to each variable in V, $OV \subseteq V$ is the set of observation variables, and $O : \mathcal{P} \rightarrow OV$ is a bijective function that associates an observation variable to a proposition. The observation variable $O(A)$ provides the truth value for A. In V there is usually a variable denoting the origin time, set to 0. In this paper this variable will be denoted by x_0. Thus, in CTPs, variables are labelled with conjunctions of literals, and the truth value of such labels are used to determine whether variables represent events that are part of the temporal problem. In this paper we consider only CTPs where E contains only STP constraints. In a CTP, for a variable to be executed, its associated label must be true. The truth values of the propositions appearing in the labels are provided when the corresponding observation variables are executed. The constraint graph of a CTP is a graph where nodes correspond to variables and edges to constraints. Nodes v is labeled with $L(v)$ and edge c is labeled with the interval of constraint c. Labels equal to $true$ are not specified. An *execution scenario* s is a conjunction of literals that partitions the set of variables in two subsets: the subset of the variables that will be executed because their label is true given s, and the subset of the other variables, that will not be executed. SC is the set of all scenarios. Given a scenario s, its *projection*, $Pr(s)$, is the set of variables that are executed under s and all the constraints between pairs of them. $Pr(s)$ is a non-conditional temporal problem.

Given a scenario s and a schedule T, for each variable v we can determine the truth values of the observations performed before time $T(v)$. The set of these outcomes will be called *observation history* of v w.r.t schedule T and scenario s, and will be written $H(v, s, T)$.

Figure 1 shows an example inspired from [13]. The example is about a plan to go skiing at station $Sk1$ or $Sk2$, depending on the condition of road R. Station $Sk2$ can be reached in any case, while station $Sk1$ can be reached only if road R is accessible. If $Sk1$ is reachable, we choose to go there. Moreover, temporal constraints limit the arrival times at the skiing station. The condition of road R can be assessed when arriving at village W. In the figure, variables XY_s and XY_e represent the start and the end time for the trip from X to Y. Node $O(A)$, where A = "road R is accessible" is HW_e. There are two scenarios, A on variables $\{x_0, HW_s, HW_e, WSk1_s, WSk1_e\}$ and $\neg A$ on variables $\{x_0, HW_s, HW_e, WSk2_s, WSk2_e\}$.

In CTPs there are three different notions of consistency depending on the assumptions made about the availability of observation information:

– *Strong Consistency (SC).* Strong consistency applies when no information is available. A CTP is strongly consistent if there is a fixed way to assign values to all the variables so that all constraints are satisfied independently of the observations. A CTP is strongly consistent if and only if its non-conditional counterpart is consistent. Therefore, an algorithm to check SC of a CTP takes the same time as checking the consistency of an STP, which is polynomial.

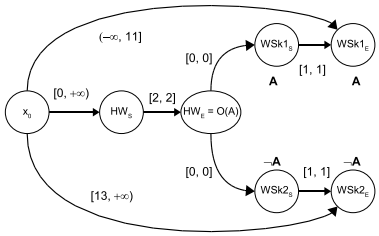


Fig. 1. Example of Conditional Temporal Problem.

- *Weak Consistency (WC).* Weak consistency applies when all information is available before execution. A CTP is weakly consistent if the projection of any scenario is consistent. Checking WC is a co-NP complete problem [13]. A brute force algorithm to check WC can check the consistency of all projections, possibly exploiting equivalent scenarios and shared paths.
- *Dynamic Consistency (DC).* Dynamic consistency (DC) assumes that information about observations becomes known during execution. A CTP is dynamically consistent if it can be executed so that the current partial solution can be consistently extended independently of the upcoming observations.

The CTP depicted in Figure 1 is not SC since, if A is true, the event HW_e must occur before 10am, instead, if $\neg A$ is true, it must happen after 12pm. However, there is a viable execution strategy, therefore the CTP is WC. Finally, being at village W is a precondition for the observation of proposition A; thus it is not possible to determine the value of A "in time" in order to schedule the departure, and the problem is not DC.

3 Fuzzifying CTPs

The conditional nature of CTPs is enclosed in the variables' labels, whose truth value enables or disables the presence of variables in the problem. Such labels indeed act as rules that select different execution paths, which, given variable v and its label $L(v)$, can be written as follows: IF $L(v)$ THEN EXECUTE (v).

The idea of fuzzifying such kind of rules has been already taken into consideration, for example in the field of fuzzy control [7, 3]. In a general study of such rules [5], both the premise and the consequence of the rule have been equipped with truth degrees associated with them. We will do the same for CTP's rules.

In our case, however, these two degrees have different meanings: the degree of the premise is used to establish if the variable should be executed, and therefore provides a truth value; the degree of the consequence, instead, can be considered as a preference on the execution of the variable.

For this reason, among the four types of fuzzy rules proposed in [5], we have chosen what are called *possibility rules*, which provide weighted conclusions.

Boolean propositions were justified in CTPs, where labels were evaluated in a crisp way, but in CTPPs they would reduce the expressiveness of the fuzzy rules; for this

reason CTPPs will be equipped with a set $\mathcal{P}$ of fuzzy atomic propositions and a set of fuzzy literals $\mathcal{Q} = \{p_i : p_i \in \mathcal{P}\} \cup \{\neg p_i : p_i \in \mathcal{P}\}$ which are mapped to values from $[0, 1]$ by an interpretation function.

Definition 1 (Interpretation function). *An interpretation function is a function* $deg : \mathcal{W} \subseteq \mathcal{Q} \rightarrow [0, 1]$, *where* $l \in \mathcal{W}$ *iff* $\neg l \in \mathcal{W}$ *and* $\forall l \in \mathcal{W}$, $deg(\neg l) = 1 - deg(l)$.

The rules we will use to fuzzify CTPs are of the form

$$\text{IF } pt(L(v), deg) > \alpha \text{ THEN EXECUTE } (v) : cp(pt(L(v, deg),))$$

where $L(v) \in \mathcal{Q}^*$ is the "fuzzy" label of variable v, deg is an interpretation function, function pt gives the truth degree of $L(v)$ given deg, and cp is the preference function associated with the consequence. The set of all "truth-preference" fuzzy rules will be named $\mathcal{FR}$.

To interpret a conjunction of fuzzy literals, given an interpretation deg, it is natural to take their minimum degree, as usual in conjunctive fuzzy reasoning. Thus function $pt : \mathcal{Q}^* \rightarrow [0, 1]$ will be the min operator.

Definition 2 (pt function). *Let* $L(v) = \wedge_{i=1,\ldots,n} l_i$, $v \in V$, $l_i \in \mathcal{W} \subseteq \mathcal{Q}$, *and* $deg : \mathcal{W} \rightarrow [0, 1]$, *then* $pt(L(v), deg) = min\{deg(l_1), \ldots, deg(l_n)\}$.

For example, a fuzzy proposition A representing sentence "It is hot" can be true with different degrees. We could say it is true with degree $deg(A) = 0.4$ if the outside temperature is mild, and with degree $deg(A) = 0.8$, if the outside temperature is above $80F$. Similarly a fuzzy proposition B representing sentence "I'm thirsty" can reasonably have different truth degrees. We can imagine attaching to a variable v, representing the time at which we go buy a cold drink, label $L(v) = AB$. This will allow us to construct a rule for v which will activate variable "get cold drink" only if the heat level or the thirst are above a given threshold.

Since we will always use the above function pt, each rule can be characterized by its threshold and its preference function. Thus we will sometimes denote a rule via the notation $r(\alpha, cp)$.

Each fuzzy rule states that variable v is part of the problem if value $pt(L(v), deg)$ is greater than the threshold α. Moreover, the consequence specifies the preference associated with the execution of v. In general, such a preference can depend on the truth degree of the premise and on the time at which v is executed. Therefore, it is reasonable to define $cp : [0, 1] \rightarrow (\Re^+ \rightarrow [0, 1])$, that is, as a function which takes in input the truth degree of the premise, i.e., $pt(L(v), deg)$, and returns a function which, in turn, takes in input an execution time and returns a preference in $[0, 1]$.

In other words, function cp allows us to give a preference function on the execution time of v which depends on the truth degree of the label of v. However, this also allows us to model situations where the preference function for the activation of v is independent of the truth degree of the premise, as a special case in which function cp has type $cp : \Re^+ \rightarrow [0, 1]$. This restricted kind of rules will be named r-cp.

In CTPs, a variable without a label implicitly has a label with value true. Similarly, in the fuzzy extension we consider, any variable whose associated rule is not specified has the following implicit one: IF $true$ THEN EXECUTE $(v) : 1$. This means that

variable v is always present in the problem, and its execution has preference 1 independently of the execution time.

Definition 3 (CTPP). *A CTPP is a tuple* $< V, E, L, R, OV, O, \mathcal{P} >$ *where:*

- $\mathcal{P}$ *is a finite set of fuzzy atomic propositions with truth degrees in* $[0, 1]$*;*
- V *is a set of variables;*
- E *is a set of soft temporal constraints between pairs of variables* $v_i \in V$*;*
- $L : V \rightarrow \mathcal{Q}^*$ *is a function attaching conjunctions of fuzzy literals* $\mathcal{Q} = \{p_i : p_i \in P\} \cup \{\neg p_i : p_i \in P\}$ *to each variable* $v_i \in V$*;*
- $R : V \rightarrow \mathcal{FR}$ *is a function attaching a "truth-preference" fuzzy rule* $r(\alpha_i, cp)$ *to each variable* $v_i \in V$*;*
- $OV \subseteq V$ *is the set of observation variables;*
- $O : \mathcal{P} \rightarrow OV$ *is a bijective function that associates an observation variable to each fuzzy atomic proposition. Variable* $O(A)$ *provides the truth degree for* A*.*

As explained above, the execution of a variable $v \in V$ depends on the evaluation of the fuzzy rule associated with it. A value assigned to a variable $v \in V$ represents the time at which the action represented by v is executed; this value will be also written as $T(v)$. If v is an observation variable it also represents the time at which the truth degree of the observed proposition is revealed.

Once a CTPP is defined, it is advisable to check statically if the information on labels and rules is consistent similarly to what is done in CTPs. In particular, if a variable v is executed, all the observation variables of the propositions in its label $L(v)$ must have been executed before v. In CTPs this is tested by checking if for each $v \in V$ and for each proposition $A \in L(v)$, $L(v) \supseteq L(O(A))$ and $T(O(A)) < T(v)$, where $O(A)$ is the observation node of proposition A.

In the fuzzy case, where conjunction is replaced by minimum and the truth values of the propositions are in $[0, 1]$, $L(v) \supseteq L(O(A))$ has to be augmented with the condition that the threshold in the rule associated with $O(A)$ should not be lower than the threshold of the rule associated to v. More formally:

Definition 4 (Structural Consistency). *Let* v *be a variable of a CTPP and* $L(v)$ *its label. A CTPP is* ***structurally consistent*** *if each observation variable, say* $O(A)$*, which evaluates a fuzzy proposition* $A \in L(v)$*, is such that* $L(O(A)) \subseteq L(v)$ *and* $\alpha \geq \beta$*, where* $R(v) = r(\alpha, cp)$ *and* $R(O(A)) = r(\beta, cp')$*.*

Checking the structural consistency of a CTPP can be performed in $O(|V|^2)$ since to establish the consistency of the label of a variable at most $O(|V|)$ labels (and thresholds) must be considered.

The definitions of scenario, projection, schedule and strategy are analogous to the classical counterparts.

Definition 5 (Scenario). *Given an CTPP* P *with a set of fuzzy literals* $\mathcal{Q}$*, a* ***scenario*** *is an interpretation function* $s : \mathcal{W} \rightarrow [0, 1]$ *where* $\mathcal{W} \subseteq \mathcal{Q}$ *that partitions the variables of* P *in two sets: set* V_1*, containing the variables that will be executed and set* V_2 *containing the variables which will not be executed. A variable* v*, with associated rule* $r(\alpha, cp)$*, is in* V_1 *iff* $pt(L(v), s) \geq \alpha$*, otherwise it is in* V_2*.* $S(P)$ *is the set of all scenarios of* P*.*

Definition 6 (Partial scenario). *A **partial scenario** is an interpretation function $s : \mathcal{W} \rightarrow [0,1]$ where $\mathcal{W} \subseteq \mathcal{Q}$ that partitions the variables of the CTPP in three sets: set V_1, containing the variables that will be executed, set V_2 containing the variables which will not be executed and set V_3 containing the variables the execution of which cannot be decided given the information provided by s. A variable v, with associated rule $r(\alpha, cp)$ and label $L(v)$, is in V_3 iff $L(v) \supset \mathcal{W}$, is in V_1 iff $pt(L(v), s) \geq \alpha$, otherwise it is in V_2.*

Since a scenario chooses a value for each fuzzy literal, it determines which variables are executed and also which preference function must be used for their execution. This means that a scenario projection must contain the executed variables, the temporal constraints among them, and the information given by the preference function of each of the executed variables. This information can be modelled by additional constraints between the origin of time and the executed variables.

Definition 7 (Constraints induced by a scenario). *Given a (possibly partial) scenario s and a variable v executed in s, consider its associated rule $r(\alpha, f) = R(v)$. The constraint induced by this rule in scenario s is the soft temporal constraint $cst_s(v)$ defined on variables x_0 and v by $(0 \leq v - x_0 < +\infty)$ with associated constraint preference function $f(min_{A \in L(v)} s(A))$. The constraints induced by scenario s are all the constraints induced by variables executed in s, that is, $U(s) = \{cst_s(v), v \text{ executed in } s\}$.*

Definition 8 (Scenario projection). *Given an CTPP P and a scenario (or partial scenario) s of P, its **projection** $Pr(s)$ is the STPP obtained by considering the set of variables of P executed under s, all the constraints among them, and the constraints in $U(s)$. Two scenarios are **equivalent** if they induce the same projection.*

Definition 9 (Schedule). *A **schedule** $T : V \rightarrow \Re^+$ of a CTPP P is an assignment of execution times to the variables in V. Given a scenario s and a schedule T, the preference degree of T in s is $pref_s(T) = min_{c_{ij} \in Pr(s)} f_{ij}(T(v_j) - T(v_i))$, where f_{ij} is the preference function of constraint c_{ij} defined over variables v_i and v_j. We indicate with $\mathcal{T}$ the set of all schedules.*

Given a CTPP P an execution strategy $St : S(P) \rightarrow \mathcal{T}$ is a function from scenarios to schedules.

Figure 2 shows an example of CTPP that extends the CTP in Figure 1. There are three skiing stations: $Sk1$, $Sk2$ and $Sk3$. A represents the fuzzy proposition "there is no snow"; station $Sk1$ is the least accessible, so it is reachable only if A is at least 0.8; on the other hand, station $Sk3$ has the most reliable roads, so it is accessible when A is above 0.3; station $Sk2$ has intermediate reachability conditions, so it is accessible for values of A above 0.5. At the same time, however, the higher the snow, the more preferable it is to go skiing. For this reason, the *cp* functions of the rules are "inversely" proportional to the truth degree of observation A. For example, this function could be $cp(x) = (1 - x)$. The two temporal constraints of the original example from x_0 to $WSk1_e$ and to $WSk3_e$ have been fuzzyfied by using trapezoidal preference functions. The preference functions for the other constraints have been omitted, meaning that they

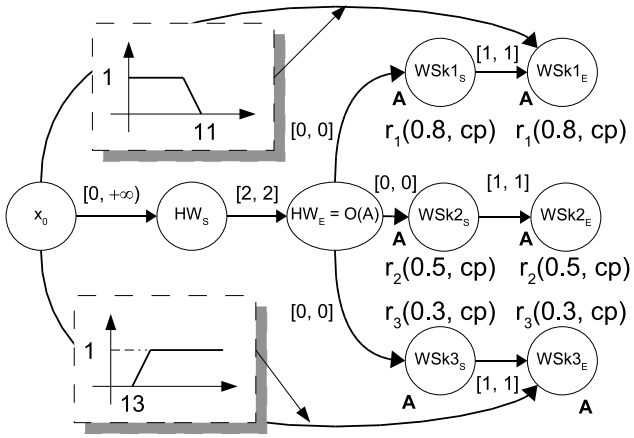


Fig. 2. Example of Conditional Temporal Problem with Preferences.

are constant functions always returning 1. In this example there are four distinct scenarios, given by $s_1(A) = 1$, $s_2(A) = 0.8$, $s_3(A) = 0.5$, and $s_4(A) = 0.3$. Thus projection $Pr(s_1)$ is the STPP defined on variables $x_0, HW_s, HW_e, WSk1_s, WSk1_e$, projection $Pr(s_2)$ is the STPP over variables $x_0, HW_s, HW_e, WSk2_s, WSk2_e$, projection $Pr(s_3)$ is the CTPP over Sx_0, HW_s, HW_e, $WSk3_s, WSk3_e$, and projection $Pr(s_4)$ is the STPP over x_0, HW_s, HW_e.

4 Consistency notions in CTPPs

Consistency notions in CTPPs are analogous to the ones in CTPs. However, we now have to consider also the preferences. There are again three notions of consistency depending on the assumptions made about the availability of the uncertain information.

Definition 10 (α**-Strong Consistency).** *A CTPP is α-strongly consistent if there is a viable execution strategy St such that, for every scenarios s_1 and s_2, and variable v executed in both,*

1. $[St(s_1)](v) = [St(s_2)](v)$*;*
2. *the global preference of $St(s_1)$ and of $St(s_2)$ is at least α.*

In words, to be α-strong consistent, we must have a schedule that satisfies all the constraints independently of the observations, giving a global preference greater than or equal to α. This is the strongest consistency notion since it requires the existence of a single schedule that gives preference at least α in every scenario. On the contrary, we can just require the existence for every scenario of a schedule (possibly a different one for different scenarios) that has a preference of at least α given the corresponding projection. This notion is that of α-weak consistency.

Definition 11 (**α-Weak Consistency**). *A CTPP Q is said* ***α-weakly consistent*** *(α-WC) if, for every scenario $s \in S(Q)$, $Pr(s)$ is consistent in the STPP sense with preference degree at least α.*

The above definitions are at the two extremes w.r.t. assumptions made on which events will be executed: α-SC assumes no knowledge at all, while α-WC assumes the scenario is given. A notion consistency which lies in between is α-dynamic consistency which assumes that the information on which variables are executed becomes available during execution in an on-line fashion. In order to define it, we first need to say when a partial scenario and a scenario are consistent.

Definition 12 (**Cons(s,w)**). *Given a CTPP P and scenario s we say a partial scenario w is consistent with s, written $Con(s, w)$ if: STPP $Pr(w)$ is a sub-problem of STPP $Pr(s)$, in the sense that the set of variables (resp. constraints) of $Pr(w)$ is a subset of the set of variables (resp. constraints) of $Pr(s)$ and no variable executed given s is not executed given w.*

This definition extends the one given in the classical case, where it is sufficient to say that a partial assignment is consistent with a scenario if the variables executed by the partial assignment are a subset of those executed by the scenario. We will use this notion in the definition of α-Dynamic Consistency, to express when at a given time the set of observations collected at that time is consistent with a scenario.

Definition 13 (**α-Dynamic Consistency**). *A CTPP is said α-dynamically consistent if there exists a viable execution strategy St such that $\forall v$ and for each pair of scenarios s_1 and s_2 $[Con(s_2, H(v, s_1, St(s_1))) \vee (Con(s_1, H(v, s_2, St(s_2))))] \Rightarrow [St(s_1)](v) = [St(s_2)](v)$ and the global preferences of $St(s_1)$ and $St(s_2)$ are at least α.*

In words, a CTPP is α-DC if for every variable v, whenever two scenarios (s_1 and s_2) are not distinguishable at the execution time for v ($Con(s_2, H(v, s_1, St(s_1))) \vee (Con(s_1, H(v, s_2, St(s_2))$, there is an assignment to v ($[St(s_1)](v) = [St(s_2)](v)]$) which can be extended to a complete assignment which in both scenarios will have preference at least α.

It is easy to see that, as for CTPs, α-SC $\Rightarrow$ α-DC $\Rightarrow$ α-WC. Moreover, given $\alpha \in [0, 1]$, if an CTPP is α-SC/DC/WC then it is β-SC/DC/WC $\forall \beta \leq \alpha$.

In what follows we consider a property which is common to all three the consistency notions. In order to do so we consider a subclass of CTPPs characterized by a special type of truth-preference rules. We will then show that the consistency of general CTPPs is equivalent to the consistency of a related problem in such a subclass.

CTPPs with restricted rules. We start by considering a simplified case, that is, when the preference functions of the rules are independent of the truth degree of the label $pt(L(v), deg)$. In such a case, given rule $r(\alpha, f)$, we assume that f is an r-*cp* function. CTPPs with such a restriction will be denoted by R-CTPPs.

The preference information given by f can be equivalently expressed by adding a constraint between the origin of time x_0 and the variable to which rule r is associated. More precisely, the constraint induced by v is the soft temporal constraint $cst(v)$ defined

on variables x_0 and v by $(0 \leq v - x_0 < +\infty)$ with associated preference function $min_{\alpha \in [0,1]} f(\alpha)$. The constraints induced by a whole CTPP Q are all the constraints induced by the variables of Q, that is, $U(Q) = \{cst(v), v \text{ variable of } Q\}$.

In the specific case of an R-CTPP Q, the preference function of each constraint in $U(Q)$ will just be $f(\alpha)$, since in this case f does not depend on the truth value of the propositions in the premise of the rule.

Theorem 1 *Given a CTPP $Q =< V, E, L, R, OV, O, \mathcal{P} >$, let us define a function R' from R as follows: if $R(v) = r(\alpha, f)$, then $R'(v) = r(\alpha, f')$ where $f' = min_{\beta \in [0,1]} f(\beta)$. Then $Q' =< V, E, L, R', OV, O, \mathcal{P} >$ is an R-CTPP. Moreover, Q is α-SC/DC/WC if and only if Q' is α-SC/DC/WC.*

5 Testing consistency of CTPPs

Thanks to Theorem 1, when testing the consistency of a CTPP we can restrict ourselves to testing the consistency of its related R-CTPP without loss of generality.

5.1 Testing α-SC

The algorithm we propose to test the α-SC of an R-CTPP is based on the correspondence of the α-SC of the R-CTPP and the consistency preference degree of a related STPP.

Theorem 2 *Given an R-CTPP $M =< V, E, L, R, OV, O, \mathcal{P} >$, let $E' = E \cup U(M)$. Then M is α-strongly consistent if and only if the STPP $< V, E' >$ is consistent with preference degree α.*

Theorem 2 relates the α-SC of an R-CTPP to the consistency level of an STPP. This allows us to check the α-SC of an R-CTPP by just constructing the appropriate STPP and then finding its best level of consistency. This will give us the highest level α at which the R-CTPP is α-SC. Since, under some tractability assumptions, solving a fuzzy STPP can be done in polynomial time [6], $U(Q)$ contains $O(|V|)$ constraints, the procedure takes polynomial time.

5.2 Testing α-Weak Consistency

In classical CTPs, the problem of checking WC is a co-$\mathcal{NP}$ complete problem [13]. Therefore, being CTPPs an extension of CTPs, we cannot expect to do better. The classical algorithm to test the WC of CTPs checks the consistency of all complete scenarios by identifying a set of labels LS that covers all the scenarios [12]. As seen in the example in Figure 2, the scenarios of a CTPP are determined not only by the labels used in the problem, but also by the thresholds levels. However, in the case of R-CTPPs, the definition of equivalence between scenarios collapses to that for CTPs, that is, two scenarios are equivalent iff they induce the same partition of the variables. In fact, in R-CTPPs the preference on the induced constraint is independent of the value of the

observation in the head of the corresponding rule. Thus the projection of the scenario is fully specified by the set of executed variables.

We first define for each literal $l \in \mathcal{Q}$ an auxiliary set $M(l)$ that contains the set of the threshold levels of truth-preference rules defined on labels containing l. More precisely: $M(l) = \{\alpha_i : \exists v \in V$ with $R(v) = r(\alpha_i, cp) \wedge l \in L(v)\} \cup \{1\}$.

Given set $M(l)$ for each literal l, we consider scenarios mapping each literal l into a value in $M(l)$.

Definition 14 (Meta-scenario). *Given a CTPP P with set of fuzzy literals $\mathcal{Q}$ a meta-scenario is an interpretation function $ms : (\mathcal{W} \subseteq \mathcal{Q}) \rightarrow \cup_{l \in \mathcal{W}} M(l)$ such that $ms(l) \in M(l)$, $\forall l \in \mathcal{W}$. We will denote the set of meta-scenarios as $MS(P) \subset S(P)$.*

Given the equivalence relation defined on R-CTPP scenarios, every scenario $s \in S(P) \setminus MS(P)$ is equivalent to a meta-scenario $ms \in MS(P)$.

Theorem 1. *Given an R-CTPP P, $\forall s \in S(P)$, $\exists ms \in MS(P)$ s.t. $Pr(s) = Pr(ms)$.*

In particular, from the above theorem we can immediately deduce that a R-CTPP is α-WC if and only if all projections of meta-scenarios are consistent with optimal preference level at least α. However, two meta-scenarios in $MS(P)$ can be equivalent. In order to further reduce the set of projections to be considered, we apply a procedure similar to that proposed in [13], in order to find a minimal set of meta-scenarios containing only one meta-scenario for each equivalence class. The procedure we propose is Algorithm 1.

Algorithm 1: FuzzyScenarioTree

Input: $\mathcal{SL}$: proposition set, s: partial meta-scenario, $ExecVars$: set of variables, PV: set of sets of variables, MS: set of meta-scenarios
Output: MS': set of sets of assignments to propositions
begin
  if $SL = \emptyset$ **then**
    return MS
  $H \leftarrow choose(\mathcal{SL})$;
  $\mathcal{SL} \leftarrow \mathcal{SL} - \{H\}$;
  foreach $\alpha \in M(H)$ `/* in increasing order */`
  do
    $s \leftarrow s \cup \{H = \alpha\}$;
    $ExecVars \leftarrow ConsVars(ExecVars, s, P)$;
    if $ExecVars \notin PV$ **then**
      if $\mathcal{SL} = \emptyset \vee ExecVars = \emptyset$ **then**
        $PV \leftarrow PV \cup \{ExecVars\}$;
        $MS \leftarrow MS \cup s$;
      else
        FuzzyScenarioTree($\mathcal{SL}$, s, $ExecVars$, MS);
end

Algorithm 1 takes in input a set of propositions $\mathcal{SL}$, a current partial meta-scenario s, the set $ExecVars$ of variables which can be executed given the information in s, the set PV containing the sets of executed variables already considered, and, finally, the set MS of meta-scenarios selected so far. In output, it gives set of meta-scenarios MS'. Algorithm 1 first considers if the set of propositions $\mathcal{SL}$ is empty and, if so, it returns the current set of meta-scenarios MS. Otherwise, it chooses (in some pre-fixed order) proposition H and then removes it from $\mathcal{SL}$. Next, for each threshold α (in increasing order) in the set $M(H)$, it extends the current meta-scenario with assignment $H = \alpha$ and computes the set of variables $ExecVars$ which are or could be executed given the information in s. In more detail, procedure $ConsVars$ takes in input a set of variables X, a partial meta-scenario w, and a CTPP P, and returns the subset of variables of X containing only variables that in P are associated with a rule whose head is not false given w (set $V_1 \cup V_3$ according to the notation of Definition 6).

If set $ExecVars$ has not been considered before (that is, it is not contained in set PV) then, if either all the propositions in $\mathcal{SL}$ have been considered or $ExecVars$ is empty, then $ExecVars$ is added to set PV and the set of meta-scenarios MS is updated with the new meta-scenario found s. Otherwise, if neither of the above sets are empty the search is carried on recursively.

In order to find a minimal set of meta-scenarios of an R-CTPP P with proposition set $\mathcal{P}$, Algorithm 1 is called with $\mathcal{SL} = \mathcal{P}$, $s = nil$[1] $ExecVars = V$, $PV = \emptyset$, $MS = \emptyset$.

The key idea of the algorithm is that as we extend a partial scenario the set of variables that could be executed can only shrink. Moreover, since for each proposition H the thresholds in $M(H)$ are considered in increasing order, when a set of executed variables is found, all its subsets have already been considered and thus if such a set is already in PV the search can avoid the recursive call.

Theorem 2. *Consider an R-CTPP P with proposition set $\mathcal{P}$. Let MS' be the set of meta-scenarios returned by Algorithm 1 when called on $\mathcal{SL} = \mathcal{P}$, $s = nil$, $ExecVars = V$, $PV = \emptyset$, $MS = \emptyset$. Then:*

- $\forall s \in MS'$, $s \in MS(P)$;
- $\forall s' \in MS(P)$, $\exists s \in MS'$ *such that* $Pr(s') = Pr(s)$;
- $\forall s, s' \in MS'$, $Pr(s) \neq Pr(s')$;

The complexity of Algorithm 1 is $O(\Pi_{H \in\ mathcalSL}|M(H)|)$ since, in the worst case the algorithm explores the whole set of meta-scenarios, of size $\Pi_{H \in \mathcal{SL}}|M(H)|$.

Example 1. Consider the following R-CTPP with four variables v_1, v_2, v_3, v_4 whose associated rules are $R(v_1) = r(0.3, \text{IF } A > 0.3 \text{ THEN EXECUTE } v_1 : 1)$, $R(v_2) = r(0.5, \text{IF } A > 0.5 \text{ THEN EXECUTE } v_2 : 1)$, $R(v_3) = r(0.2, \text{IF } AB > 0.2 \text{ THEN EXECUTE } v_3 : 1)$, and $R(v_4) = r(0.5, \text{IF } AB > 0.5 \text{ THEN EXECUTE } v_4 : 1)$. In this case $M(A) = \{0.2, 0.3, 0.5, 1\}$ and $M(B) = \{0.2, 0.5, 1\}$. This problem has 12 meta-scenarios, while the minimal set is $\{\{A = 0.2\}, \{A = 0.3, B = 0.2\}, \{A = 0.5, B = 0.2\}, \{A = 0.5, B = 0.5\}, \{A = 1, B = 0.2\}, \{A = 1, B = 0.5\}, \{A = 1, B = 1\}\}$.

[1] We write $s = nil$ meaning the function with the empty domain, that is, to model a partial scenario in which no proposition is assigned.

The algorithm we propose to test α-WC of a R-CTPPcomputes a minimal set of meta-scenarios applying Algorithm 1 and for each such meta-scenario ms it checks if the corresponding projection $Pr(ms)$ is consistent at level α. If the preference functions are semi-convex, in order to test this it is sufficient to test whether the STP obtained from $Pr(s)$ via its α-cut (that is considering for each constraint the sub-interval containing elements mapped into a preference $\geq \alpha$) is consistent.

If the preference functions are semi-convex the co-problem of α-WC is NP-complete since it coincides with deciding if there is an inconsistent STP obtained via the α-cuts. Thus in such a case testing α-WC is co-NP-complete.

5.3 Testing α-Dynamic Consistency

In [13] the DC of a CTP is checked by transforming the CTP into a Disjoint Temporal Problem (DTP) [11] obtained from the union of the STPs corresponding to the projections of the scenarios of the CTP and some additional disjunctive constraints. A CTP is DC if, whenever at certain point in time a given variable must be executed, and it is not possible to distinguish in which scenario we are, there is a value to assign to such a variable which will be consistent with all the possible scenarios that can evolve in future. This means that all the variables representing the same CTP variable in the projections either are constrained to be after observations which allow to distinguish the scenario univocally (and thus can be executed independently of each other) or they must be assigned the same value whenever observation variables do not allow to distinguish the scenarios. This is modelled by adding to the STP, obtained by the union of all the projections of the CTP, a specific set of disjunctive constraints (called CD). This makes the STP become a DTP (see [13] for more details).

Since in R-CTPPs executing a variable at the same time in different scenarios gives the same preference, the reasoning above can be applied directly. In fact, in terms of synchronization only the temporal order matters.

Theorem 3 *Given an R-CTPP $Q =< V, E, L, R, OV, O, \mathcal{P} >$, let $D = \langle V', E' \rangle$ be the fuzzy DTP with $V' = (\bigcup_{Pr(s)=(V,E),s\in MS'} V)$ and $E' = (\bigcup_{Pr(s)=(V,E),s\in MS'} E) \cup CD$. Then Q is α-dynamically consistent if and only if D is consistent with preference degree α.*

Theorem 3 allows us to define an algorithm which, given in input an R-CTPP, tests if it is α-DC. Such an algorithm first computes the minimal set of meta-scenarios by applying Algorithm 1. Next, it tests if the DTPP obtained taking the union of the all the STPPs corresponding to projections of meta-scenarios in the minimal set, and adding the CD constraints, is consistent with optimal preference level α. Thus the complexity of checking α-DC is the same as that of solving a fuzzy DTPP; we recall that efficient algorithms for finding the optimal preference level of Fuzzy DTPPs have been considered in [8].

6 CTPPs vs. STPPUs

It is interesting to notice that consistency in CTPs is strongly connected to controllability in STPUs. This arises from the fact that both kinds of problems are concerned

with the representation of uncertainty: STPUs model uncertainty by defining contingent constraints, while CTPs try to capture the outcomes of external events by modelling conditional executions.

We propose here a mapping from STPPUs to CTPPs that preserves the controllability/consistency of the problem. The main idea of this mapping is that, if an STPU has contingent constraints defined over finite domains, each possible value that their endpoints can assume is, in a sense, a condition which has been satisfied.

Given an STPPU $Q =< N_e, N_c, L_r, L_c >$, let $k = |L_c|$, for every soft contingent temporal constraints $l_i \in L_c$ such that $l_i =< [a_i, b_i], f_i >$ we discretize the interval $[a_i, b_i]$ and we denote the number of elements obtained with $|l_i|$ indicating such a set of elements with $\{d_{ij}, j = 1 \dots |l_i|\}$. For the sake of notation, we write $I = \{1 \dots |L_c|\}$ and, for each $i \in I$, $J_i = \{1, \dots, |l_i|\}$

Let us consider the mapping applied to a contingent constraint $l_i =< [a_i, b_i], f_i >$, defined on executable A and contingent variable C. We add $|l_i|$ observation variables, o_{ij}, and $|l_i|$ variables v_{ij}, one for each possible occurrence of C at time d_{ij} in $[a_i, b_i]$. Variable o_{ij} observes the proposition $p_{ij} =$ "$C = d''_{ij}$, while variable v_{ij} represents the actual occurrence of C at time d_{ij}.

Moreover we add a hard temporal constraint with interval $e_{ij} =< [0, 0], 1 >$ between o_{ij} and v_{ij}, and and we add a soft constraint $eo_{ij} =< [d_{ij}, d_{ij}], f_{|d_{ij}} >$ between A and o_{ij}.

Any other constraint w involving C in the STPPU is replicated $|l_i|$ times, one for each d_{ij}, obtaining constraint w_{ij} connected to the corresponding v_{ij} variable.

Definition 15. *Given an STPPU $Q =< N_e, N_c, L_r, L_c >$, where I and J_i are as above, we define the CTPP $C(Q)$ as the tuple $< V, E, L, R, OV, O, \mathcal{P} >$, where*

- *$\mathcal{P}$ is the set of fuzzy atomic propositions $\{p_{ij}, i \in I, j \in J_i\}$;*
- *$V = N_e \cup \{o_{ij}, i \in I, j \in J_i\} \cup \{v_{ij}, i \in I, j \in J_i\}$;*
- *$E = L^e_r \cup \{e_{ij}, i \in I, j \in J_i\} \cup \{eo_{ij}, i \in I, j \in J_i\} \cup \{w_{ij}, i \in I, j \in J_i\}$ where L^e_r is the set of all the requirement constraints in L_r defined only between executable variables and e_{ij}, eo_{ij}, and w_{ij} are as defined above;*
- *$L : V \to \mathcal{Q}^*$ is a function such that $L(v_{ij}) = p_{ij}$ and $true$ otherwise;*
- *$R : V \to \mathcal{FR}$ is a function defined as $R(v_{ij}) = r(0, g)$, where g is the constant function equal to $f(d_{ij})$ where is the preference function of l_i;*
- *$OV \subseteq V$ is the set of observation variables $\{o_{ij} \in I, j \in J_i\}$;*
- *$O : \mathcal{P} \to OV$ is a bijective function such that $O(p_{ij}) = o_{ij}$;*

It is possible to show that this mapping preserves the controllability/consistency notions.

Theorem 4 *Given an STPPU Q and its corresponding CTPP $C(Q)$, Q is α-strongly (resp., weakly, dynamically) controllable iff $C(Q)$ is α-strongly (resp., weakly, dynamically) consistent.*

Notice that the result above mentions α-weak controllability, which is not defined in [10], where only the stronger notion of Optimal-weak controllability is considered. However α-weak controllability can be directly obtained from the definition of Optimal weak controllability by replacing "optimal" with "$\geq \alpha$ whenever the projection has optimal preference at least α".

7 Conclusions and Future Work

We have defined Conditional Temporal Problems with fuzzy Preferences, which extend CTPs [13] both by adding preferences to the temporal constraints and by generalizing the conditions of the classic model to fuzzy rules that activate the occurrence of some events on the basis of fuzzy thresholds. Moreover, also the activation of the events is modeled with preferences. The three notions of consistency (strong, weak and dynamic) have been extended accordingly and algorithms to test them have been proposed. Complexity results show that the substantial gain in terms of expressiveness comes at a modest additional computational cost.

Future directions include: implementing and testing the algorithms on randomly generated problems and on real-life examples, extending CTPPs to preferences other than fuzzy, and integrating qualitative temporal constraints in the framework.

References

1. M. Ai-Chang, J. L. Bresina, L. Charest, A. Chase, J. Cheng jung Hsu, A. K. Jónsson, B. Kanefsky, P. H. Morris, K. Rajan, J. Yglesias, B. G. Chafin, W. C. Dias, and P. F. Maldagu. Mapgen: Mixed-initiative planning and scheduling for the mars exploration rover mission. *IEEE Intelligent Systems*, 19(1):8–12, 2004.
2. C. Combi and G. Pozzi. Task scheduling for a temporal workflow management system. In *Proc. of TIME'06*, pages 61–68, 2006.
3. E. Cox. Fuzzy fundamentals. *IEEE Spectrum*, 29(10)(10):58–61, 1992.
4. R. Dechter, I. Meiri, and J. Pearl. Temporal constraint networks. *Artificial Intelligence*, 49:61–95, 1991.
5. D. Dubois and H. Prade. What are fuzzy rules and how to use them. *Fuzzy Sets and Systems*, 84:169–185, 1996.
6. L. Khatib, P. Morris, R. A. Morris, and F. Rossi. Temporal constraint reasoning with preferences. In *IJCAI01*, pages 322–327, 2001.
7. C. C. Lee. Fuzzy logic in control systems: fuzzy controllers. *IEEE Trans. on Sys., Man and Cybern.*, 2092:404–435, 1990.
8. Bart Peintner and Martha E. Pollack. Low-cost addition of preferences to dtps and tcsps. In *Proc. of AAAI'-04*, pages 723–728. AAAI Press / The MIT Press, 2004.
9. Martha E. Pollack, Laura E. Brown, Dirk Colbry, Colleen E. McCarthy, Cheryl Orosz, Bart Peintner, Sailesh Ramakrishnan, and Ioannis Tsamardinos. Autominder: an intelligent cognitive orthotic system for people with memory impairment. *Robotics and Autonomous Systems*, 44(3-4):273–282, 2003.
10. F. Rossi, K. B. Venable, and N. Yorke-Smith. Uncertainty in soft temporal constraint problems: a general framework and controllability algorithms for the fuzzy case. *Journal of AI Research*, 27:617–674, 2006.
11. K. Stergiou and M. Koubarakis. Backtracking algorithms for disjunctions of temporal constraints. *Artificial Intelligence*, 120(1):81–117, 2000.
12. I. Tsamardinos. *Constraint-Based Temporal Reasoning Algorithms with Applications to Planning*. PhD thesis, University of Pittsburgh, 2001.
13. I. Tsamardinos, T. Vidal, and M. E. Pollack. Ctp: A new constraint-based formalism for conditional, temporal planning. *Constraints*, 8(4):365–388, 2003.
14. T. Vidal and H. Fargier. Handling contigency in temporal constraint networks. *J. of Experimental and Theoretical Artificial Intelligence Research*, 11(1):23–45, 1999.

A Genetic Algorithm Solving Method for Confident-DEA: A Generalized Approach For Efficiency Analysis With Imprecise Data.

Said Al-Gattoufi[1], Ph.D.
College of Commerce and Economics, Sultan Qaboos University, Sultanate of Oman
Gattoufi@squ.edu.om or Algattoufi@yahoo.com

Abstract

This paper proposes an extension to the existing literature in DEA, the authors call *Confident-DEA* approach. The proposed new approach involves a *bi-level convex* optimization model, and hence *NP-hard*, to which a solution method is suggested. *Confident-DEA* constitutes a generalization of DEA for dealing with imprecise data and hence a potential method for forecasting efficiency. Imprecision in data is defined as two forms, one is bounded data and the second is cardinal data. Complementing the methodology proposed by Cooper et al (1999) which provides single valued efficiency measures, *Confident-DEA* provides a range of values for the efficiency measures, e.g. an efficiency confidence interval, reflecting the imprecision in data. For the general case of imprecise data, that is a mixture of ordinal and cardinal data, a Genetic-Algorithm-based meta-heuristic is used to determine the upper and lower bounds defining the efficiency confidence interval. To the best knowledge of the authors, this is the first work combining Genetic algorithms with DEA. In both cases of imprecision, a Monte-Carlo type simulation is used to determine the distribution of the efficiency measures, taking into account the distribution of the bounded imprecise data over their corresponding intervals. Most of previous DEA works dealing with imprecise data implicitly assumed a uniform distribution. *Confident-DEA*, on the other hand, allows for any type of distribution and hence expands the scope of the analysis. The bounded data used in the illustrative examples are assumed to have truncated normal distributions. However, the methodology suggested here allows for any other distribution for the data.

1. Introduction

In the standard Data Envelopment Analysis (DEA) approach originally proposed in Charnes, Cooper and Rhodes (1978) and further detailed in Charnes and Cooper (1985), Decision Making Units (DMUs) are considered to be economic units using inputs to produce outputs. Although the inputs and outputs are usually assumed to be observable and measurable, in many real life situations these factors are not precisely known except ***(i)*** to the extent that the true values lie within prescribed bounds, and/or ***(ii)*** to satisfy certain ordinal relations. Data falling in the first category are referred as *bounded data* while the later are known as *ordinal data*. Cooper et al (1999) refers to any mixture of such kind of data with standard single valued data as *imprecise data*. We henceforth use the same terminology.

When data are imprecise, the application of the standard DEA leads to a non-linear program and the piecewise linear efficient frontier defined by that approach is not guaranteed. Moreover, ordinal data cannot be considered in the standard model.

Imprecise Data Envelopment Analysis (*IDEA*[2]), Cooper et al (1999), treats mixtures involving bounded data in addition to ordinal data and ordinal relations among the weights. However, a major conceptual criticism can be addressed to the IDEA approach. The problem lies with *the derivation of* ***single-valued*** *measures from* ***imprecise multi-valued*** *data*. *The efficiency measures should, in my opinion, reflect the imprecision in data.* In this setting, a range of values for the efficiency measure is more appropriate than a single-value. Furthermore, this range can be considered as a confidence interval for the efficiency measure and hence the name *Confident-DEA*. The spread of the efficiency interval can be interpreted as an indicator of the degree of volatility for the efficiency of the corresponding DMU. The wider is the efficiency interval, the higher is the volatility of the efficiency and hence the higher is the uncertainty about the relative performance of the corresponding DMU.

Confident-DEA extends the standard DEA to the case of imprecise data while overcoming the above mentioned weakness by producing efficiency confidence intervals. Furthermore, it compliments the *IDEA* in the sense that it allows considering stochasticity in data. In the case of bounded data, *Confident-DEA* takes into account the distribution of values of the factors within their corresponding ranges whereas IDEA implicitly assumes the uniform distribution.

[1] Corresponding author
[2] The same abbreviation e.g., IDEA, for "Interactive Data Envelopment Analysis" is used in Post and Spronk (1999). The abbreviation proposed by Cooper et al. (1999) is becoming more common in the literature.

2. The Mathematical Modeling

Given the inputs and outputs to be considered, the following model is the standard *Data Envelopment Analysis* (DEA) model :

$$
\begin{aligned}
&\underset{\mu,\omega}{Max}\, h_o = \sum_{r=1}^{r=s} \mu_r y_{ro} && (1.1)\\
&Subject\ to: && \\
&\sum_{r=1}^{r=s} \mu_r y_{rj} - \sum_{i=1}^{i=m} \omega_i x_{ij} \le 0;\ j = 1,2,\ldots,n && (1.2)\\
&\sum_{i=1}^{i=m} \omega_i x_{io} = 1 && (1.3)\\
&y_r = (y_{rj}) \in D_r^+ && (1.4)\\
&x_i = (x_{ij}) \in D_i^- && (1.5)\\
&\mu = (\mu_r) \in A^+ ,\ \mu_r \ge 0 && (1.6)\\
&\omega = (\omega_i) \in A^- ,\ \omega_i \ge 0 && (2.7)
\end{aligned}
\qquad (1)
$$

where $\boldsymbol{x}$ represents the matrix of input values for each DMU. It specifies the values of inputs used in the production process. $\boldsymbol{y}$ on the other hand represents the output matrix. It specifies, for each DMU, the values of the different outputs that result from the production process. $\boldsymbol{u}$ and $\boldsymbol{v}$ are the coefficient vectors to be determined by solving the model. $\boldsymbol{RD_r^+}$, $\boldsymbol{RD_r^-}$, $\boldsymbol{RA_r^+}$ and $\boldsymbol{RA_r^-}$ respectively represent domains for the outputs, inputs, output multipliers and input multipliers.

Real world situations often dictate data, the values of which lie within some prescribed bounds. Moreover, the data may be ordinal rather than cardinal in form, and hence known only to be satisfying certain ordinal relations. In Cooper et al (1999) these are labeled "*imprecise data*". Lastly, the *"data"* may represent the decision-maker's judgmental restrictions on the relative weights allowed to each or to some of the factors and/or multipliers. This is known in the DEA literature as the *Assurance-Region*. A specific domain for the solution search can be imposed and this is known as the *Cone-Ratio*. The general form presented above allows for all forms of data as well as all forms of restrictions on multipliers. In this paper we deal only with imprecision in Data. Restrictions are out of the scope of the work work.

In the case of imprecise data, the model presented above is not linear any longer. The standard DEA approach cannot be applied, and hence the piecewise linear efficient frontier defined that approach is not guaranteed. Formulating the basic DEA model using imprecise data leads in fact to a *non-linear optimization problem*.

The early literature dealing with imprecise data was simply devoted to extend the standard DEA for coping with ordinal data. Golany (1988) presented a model incorporating ordinal relations among the weights of the DEA model. Cook et al (1993) presented a framework for incorporating a single input within the standard DEA framework. In a follow-up work, Cook et al (1996) extended their framework to the case where more than one factor is ordinal. Kim et al (1999) developed a procedure for handling both ordinal data and weights preferences. Lee et al (2002) transformed the non-linear program obtained by considering imprecise data to a linear program, using a series of modification of variables. Cooper et al (1999) developed a unified approach to treating mixtures involving bounded data in addition to ordinal data and ordinal relations among the weights. Their approach, the *Imprecise Data Envelopment Analysis* (*IDEA*), extends the standard DEA to cope with imprecise data. In a following-up work, Cooper et al (2001a) presented an illustrative application of their unified approach. Formulating the basic DEA model using imprecise data leads to a *non-linear optimization problem*. For the linearization, IDEA proceeds in two steps, scale transformations followed by variable alterations. The transformed model has the form of a standard DEA model. The solution for the original model is obtained from that of the transformed model using the reverse variable alterations and scale transformations.

A common criticism in this respect is that these approaches do not reflect explicitly the imprecision of the data within the assessment efficiency represented by the efficiency coefficients provided. That is, the efficiency measure obtained for each DMU is single-valued regardless the data are single-valued or imprecise.

Three major critics can be addressed as regarding the *IDEA* approach. First, the authors, in order to linearize the model obtained from the application of standard DEA to imprecise data, transformed the status of data to variables. That is, the authors consider the factors of data not precisely known as variables. This leads to an optimization problem where they decide about data as well as about variables. The basic Operations Research methodology requires a clear identification and separation between the decision variables, object of decision for the optimal level they should have, and the parameters represented by the coefficient defined by the data of the problem.

Second, for a variable defined as having bounded data, the *IDEA* approach requires that for the DMU used as anchor for the scale transformation and variable alteration, that is the DMU with the highest range for the corresponding bounded variable, the range is transformed into a single-valued. If this "*approximation*" is not made, the reverse transformations to retrieve the solution for the original problem can not be performed. This reduces some of the generality of the *IDEA* approach. However, this was corrected in Cooper et al (2001b) by introducing dummy DMUs in the analysis.

Finally, the major criticism is conceptual in nature. The problem with the existing literature dealing with imprecise data is *the derivation of* ***single-valued*** *measures from* ***imprecise multi-valued*** *data*. The efficiency measure should reflect the imprecision in data and a range of values for the efficiency measure is more appropriate than a single-value. This range can be considered as a confidence interval for the efficiency measure. Later in this study, a new methodology, called *Confident-DEA,* is provided. It extends the standard DEA to the case of imprecise data and produce efficiency confidence intervals.

This paper develops a new approach, *Confident-DEA* that extends and generalizes the IDEA approach in the case of single valued and bounded cardinal data in the sense it allows imprecision in data to be reflected in the resulting efficiency measures. This is achieved by providing a range for the efficiency measures, an efficiency confidence interval and hence the name Confident-DEA, for each DMU instead of the single valued measure provided by the IDEA approach. A generalization to the case of imprecise data, an approach using a Genetic Algorithm based meta-heuristic for determining the bounds of the efficiency confidence interval is proposed in this work.

The upper bound confident efficiency interval for each DMU in the case of bounded data is obtained by solving the following model:

$$
\begin{aligned}
&\underset{x,y}{Max}\ \underset{i,u}{Max}\ h_0 = \sum_{r=1}^{r=s} \mu_r y_{ro} && (2.1)\\
&subject\ to:\\
&\sum_{r=1}^{r=s} \mu_r y_{rj} - \sum_{i=1}^{i=m} \omega_i x_{ij} \le 0;\ j = 1,2,\ldots,n && (2.2)\\
&\sum_{i=1}^{i=m} \omega_i x_{i0} = 1 && (2.3)\\
&y_r = (y_{rj}) \in D_r^+ && (2.4)\\
&x_i = (x_{ij}) \in D_i^- && (2.5)\\
&\mu = (\mu_r) \in A^+ , \mu_r \ge 0 && (2.6)\\
&\omega = (\omega_i) \in A^- , \omega_i \ge 0 && (2.7)
\end{aligned}
\qquad (2)
$$

The lower bound is determined by considering the minimization model.

These two models represent a *non-linear convex* problems and can be written in the general form of a *bilevel convex* model, discussed in greater detail by Bard (1998). In the two levels of optimization, multipliers are subjects at the lower level while the factors are subjects at the upper level. The model proceeds by determining the optimal multipliers for a given level of the factors. The general mathematical form of a b*ilevel convex* problem, where *F, G, f* and *g* are convex functions is:

$$
\begin{aligned}
&\underset{x \in X}{Max}\ F(x, \omega(x)) && (3.1)\\
&Subject\ to \quad G(x, \omega(x)) \le 0 && (3.2)\\
&\qquad \omega(x) = \underset{y \in Y}{Max}\ f(x,y) && (3.3)\\
&\qquad\quad Subject\ to\ \ g(x,y) \le 0 && (3.4)
\end{aligned}
\qquad \mathbf{(3)}
$$

The well known Max-Min problem is a special case of the general bilevel convex problem. While Jeroslow (1985) proved that the *Max-Min* problem is NP-hard, Hansen et al (1992) proved that the *linear bilevel programming problem* is *strongly NP-hard*. The significant difficulty in solving the general form, convex bilevel optimization problems, justifies the use of heuristics.

3. *Confident-DEA*: A Genetic-Algorithm-Based solving method for a mixture of data

3.1. The context for the genetic algorithm:

As mentioned previously, when imprecision in data is considered, the standard DEA model is not a linear program any longer. Furthermore, it can be seen as *bi-level convex* model, an *NP-hard* problem. This justifies finding heuristics solving methods. Our choice goes to genetic algorithm because, the high predisposition of the model to this meta heuristic.

Holland (1992) and his associates suggested initially in the sixties and seventies the basic principles of Genetic Algorithms. They are inspired by the mechanism of natural selection where stronger individuals are likely to be the winners in a competing environment. Through the genetic evolution method, an optimal, or a satisfactory, solution can be found and represented by the final winner of the genetic game. The name Genetic Algorithm originates from the analogy between the representation of a complex structure by means of a vector of components, and the idea of the genetic structure of *chromosomes* familiar to biologists. A vector, generally a sequence of 0-1 components, represents a chromosome and each component represents a *gene* that reflects a specific elementary characteristic. Manipulations made on chromosomes are called *genetic operators* and the most common are *crossover and mutation.*
The idea of Genetic Algorithm in optimization can be understood as an intelligent neighbouring random search method. While several methods using random sampling have been used, the *Genetic Algorithm* approach is more flexible and provides a new framework for a variety of problems.

The original version, Holland's version, of the Genetic Algorithm works by maintaining a population of *M* chromosomes considered as potential *parents*. Each chromosome is evaluated using a given function, and assigned a *fitness value*. Each chromosome encodes a solution to the problem and its fitness value is related to the objective function value for that solution. One parent, a chromosome, is selected on a fitness basis (the better the fitness value, the higher the chance of being chosen), while the other parent is chosen randomly. They are then *mated* by choosing a crossover point X at random, the *offspring* consists of the pre-X section from one parent followed by the post-X section of the other.

The Genetic Algorithm in general allows a *population* composed of many individuals to *evolve* under specified *selection rules* to a state that *maximizes* the *fitness*, a measure of goodness of individuals. It emulates the *survival-of-the-fittest* mechanism in nature. A mating pool is extracted from the original population of individuals or *chromosomes*. The Genetic Algorithm presumes that each chromosome, a potential candidate, can be represented by a set of parameters called *genes* and can be structured by a string of values in binary form. These selected chromosomes constitute the original set of parents.

3.2. Description of the metaheuristic proposed:

The more general case of *Confident-DEA* proposed in this section uses Genetic-Algorithm to handle a mixture of data involving ordinal, single-valued and bounded. The steps of the meta-heuristic are described in *Figure1, Figure 2* and *Figure 3*.

As any meta-heuristic, the first step is the encoding process that enables representing DMUs in the standard form for Genetic Algorithm use. For each DMU, a string of numbers is defined (continuous or discrete) representing the values of factors. For the factors presumed to be known exactly (single valued), there will be a single-value substring for each. For the bounded factors, each will be represented by a substring containing all possible values obtained from the discretization of the corresponding range. That is, the final string of numbers representing the DMU will be composed of substrings each one representing the possible value(s) for one factor. The key idea in the *Confident-DEA* approach is to represent each DMU by a set of *chromosomes*, binary strings, in which each *gene*, 1 or 0, refers to whether or not the corresponding value is assigned to the corresponding factor.

Each DMU is split into a set of chromosomes, each one representing a *virtual single-valued* alternative for the real imprecise DMU.

For the illustration of the splitting-up process and generation of virtual DMUs, let a DMU using two inputs, $\mathbf{X_1}$ and $\mathbf{X_2}$ to produce two outputs $\mathbf{Y_1}$ and $\mathbf{Y_2}$. Suppose that $\mathbf{X_1}$ and $\mathbf{Y_1}$ are presumed to be described by exact data while $\mathbf{X_2}$ and $\mathbf{Y_2}$ are described by bounded data.

Let $\mathbf{X_1 = 20}$ and $\mathbf{Y_1 = 30}$ while $\mathbf{1 < X_2 < 5}$ and $\mathbf{11 < Y_2 < 15}$.

The factor's order is arbitrarily chosen as $\mathbf{\{X_1 ; Y_1 ; X_2 ; Y_2\}}$. The semi-columns are used only for the purpose of explanation .The string of numbers representing this DMU will then be: **{20 ; 30 ; 2 3 4 ; 12 13 14 }.** The set of chromosomes representing this DMU will then be:

K_1= {1 ; 1 ; 1 0 0 ; 1 0 0}, K_2= {1 ; 1 ; 1 0 0 ; 0 1 0}, K_3= {1 ; 1 ; 1 0 0 ; 0 0 1},
K_4= {1 ; 1 ; 0 1 0 ; 1 0 0}, K_5= {1 ; 1 ; 0 1 0 ; 0 1 0}, K_6= {1 ; 1 ; 0 1 0 ; 0 0 1},
K_8= {1 ; 1 ; 0 0 1 ; 1 0 0}, K_8= {1 ; 1 ; 0 0 1 ; 0 1 0}, K_9= {1 ; 1 ; 0 0 1 ; 0 0 1}.

The factors' value of the virtual DMU represented by K_1 are:

$X_1 = 20$; $Y_1 = 30$; $X_2 = 2$; $Y_2 = 12$.

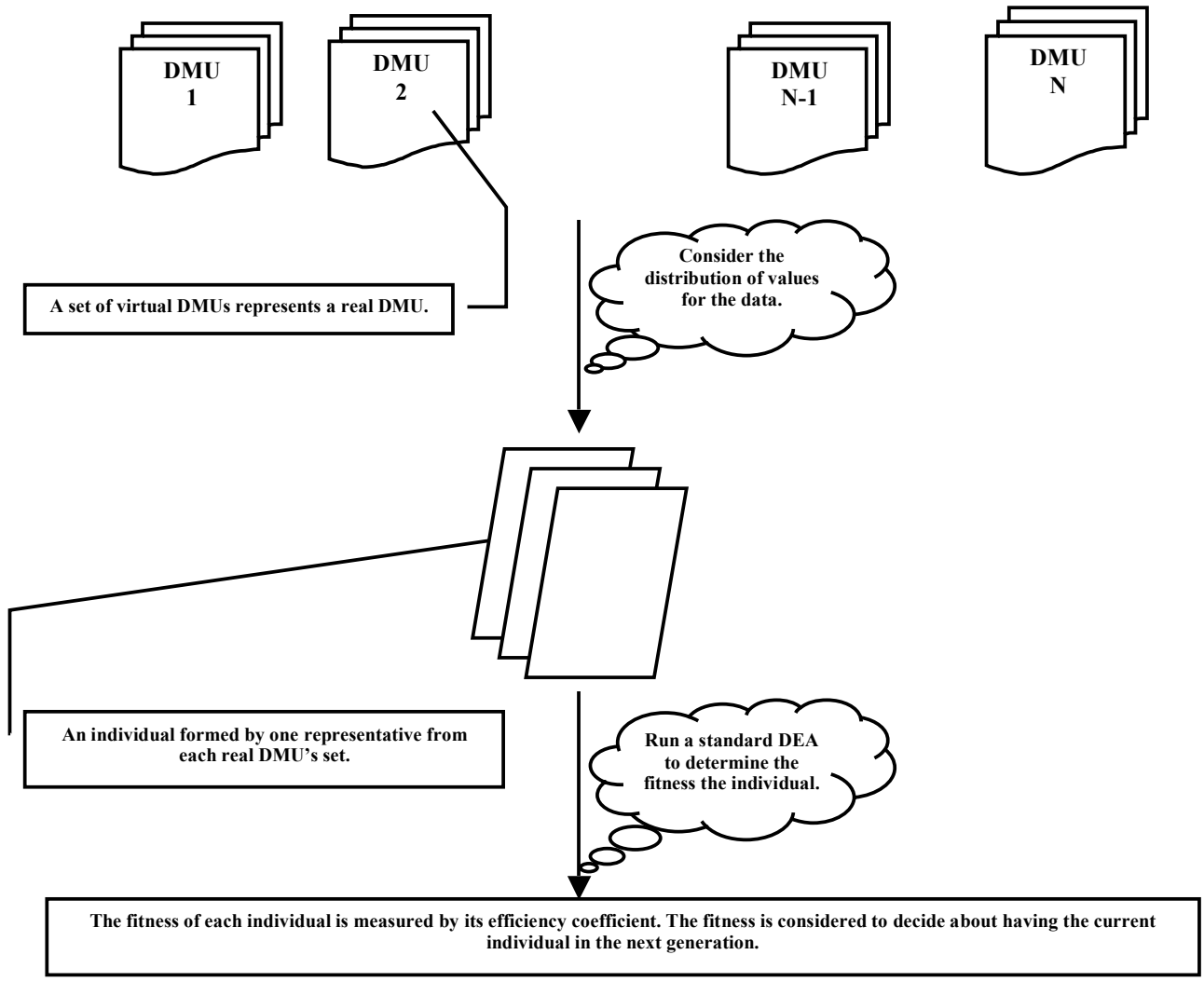


Figure 1: Splitting-up process and definition of an *"individual"*

By doing so for all DMUs, a set of virtual DMUs is obtained for each DMU. An *individual* is defined by a set of chromosomes determined by choosing, taking into account the distribution of imprecise values to make the approach stochastic, a representative from each set of virtual DMUs with exact data representing a real DMU. It is important to remark that, unlike the standard Genetic Algorithm procedure, an individual here is represented by a binary matrix rather then a binary string.

Once the encoding is realised, the Genetic Algorithm heuristic for *Confident-DEA* proceeds basically in two phases:

(i) selection of the initial population using the *Roulette Wheel* method, and

(ii) creation of the offspring using genetic modifications, to define the next generation. Multi-point crossover with high probability, around 0.9, and mutation with low probability, in the range 0.001-0.1 are the genetic modification used in the meta-heuristic. The cutting of the matrix-individual to define the crossover points is both vertical and horizontal. The size of the initial population as well as the number of iterations is set up arbitrarily at the beginning.

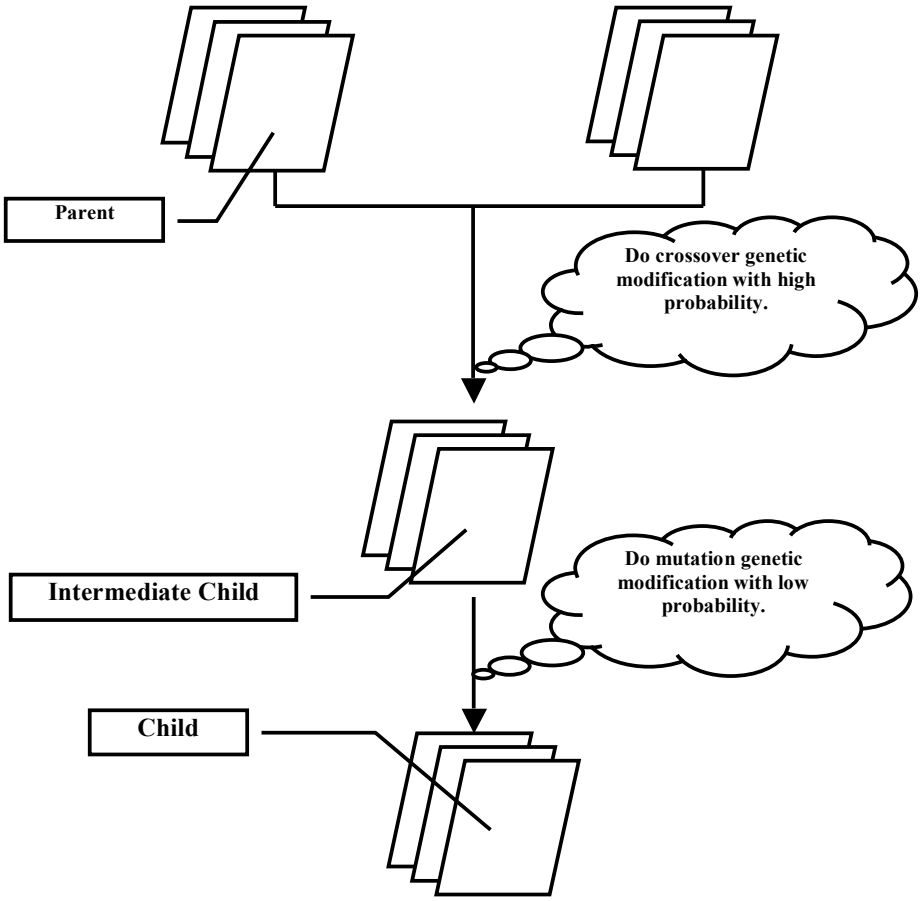


Figure 2: Genetic modifications: Crossover and Mutation

An initial population using the *Roulette Wheel* selection mechanisms is generated, and it constitutes the mating pool. An individual is a set of chromosomes, each one representing a DMU. All DMUs are represented in each individual and there is a single representative, a chromosome, of each DMU in each individual. The fitness function is the efficiency coefficient of the base-DMU.

Once the initial population is determined, the next phase is the creation of the next generation. This phase proceeds in three steps illustrated in *Figure 2*: ***(i)*** the mating of two selected individuals, considered as future parents ***(ii)*** make crossover with high probability and ***(iii)*** make mutation with low probability. All genetic modifications are decided based on the fitness of the individual determined by running a standard DEA model. The fitness measure is the efficiency coefficient of the base-DMU and it is computed for the selected individual at each step. Considering the binary matrix representing the individual, the corresponding virtual DMUs are identified. By solving the corresponding DEA model, the fitness, that is the efficiency coefficient of the base-DMU, is determined.

The process continues until a new generation is obtained. This new generation replaces the former generation and the process in initiated again. Iteration stops when the number of generations reaches the predetermined number.

The meta-heuristic proceeds in depth first, which means that all iterations are run for the first base-DMU to determine the lowest level of efficiency, then the iterations are run to determine the highest level. Once done with the first DMU, the process is iterated for the second base-DMU and so on.

Using this Genetic Algorithm based approach, summarized in *Figure 3*, an upper bound and a lower bound for the efficiency coefficient of each DMU are defined. Like any heuristic or meta-heuristic, obtaining an optimal solution is not guaranteed.

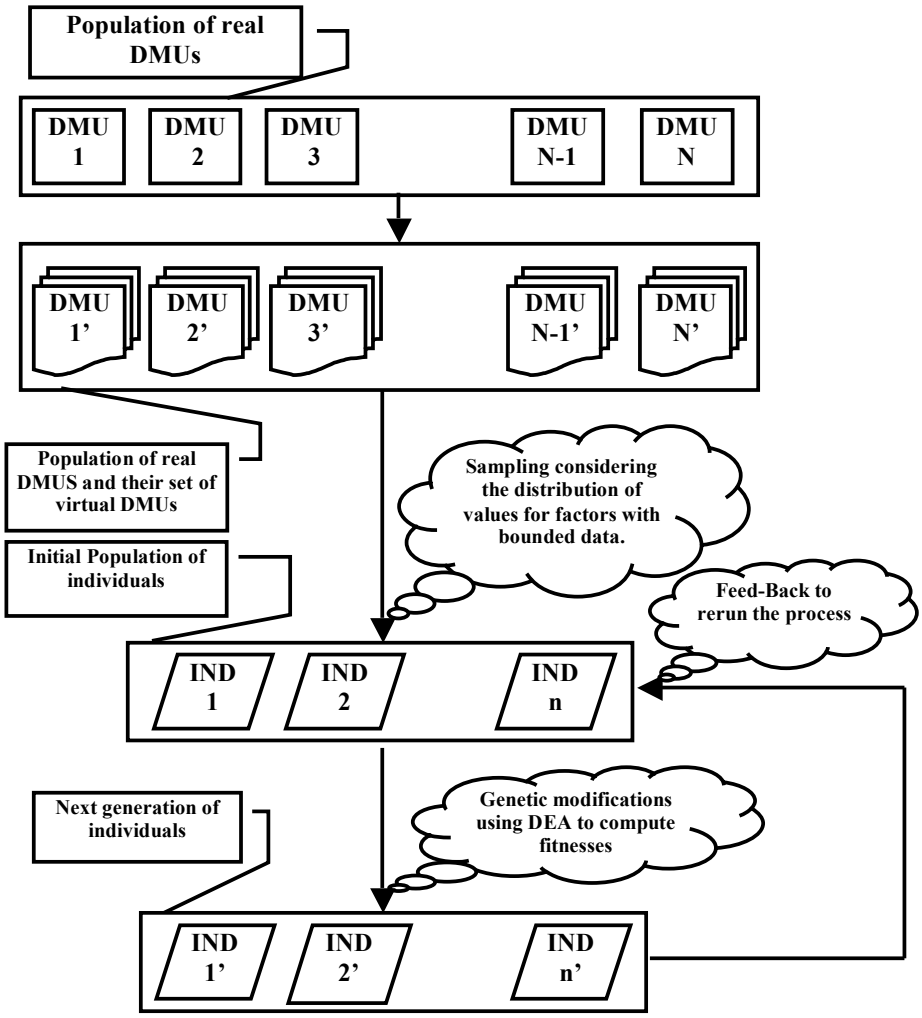


Figure 3: Methodological Contribution: Marriage of DEA with Genetic Algorithm Procedure

3.4. A Simulation-Based Component of *Confident-DEA*

The third component of *Confident-DEA* is a simulation based heuristic. It proceeds in three phases:

(i) define the individuals in the same way described for the Genetic Algorithm based heuristic,

(ii) run a standard DEA for each individual in order to determine its efficiency coefficient and

(iii) determine the confidence interval and the distribution of efficiency coefficient for each DMU by using a Monte Carlo type simulation.

Once an individual is chosen, the efficiency coefficient of each one of its virtual DMUs is computed by solving the corresponding standard DEA model. These values are stored for future comparison. In the next iteration, the coefficients obtained are compared with previous results in order to determine the minimum and the maximum efficiency level for each DMU. Once the predetermined number of iterations is reached, the output of the heuristic has three components. First, a confidence efficiency interval for each DMU is determined. Second, benchmarks for different level of efficiency are identified. Finally, the distribution for the efficiency coefficient is defined based on the frequency histogram number of hits for each predefined sub-interval of [0-1]. The interval [0-1] is in fact pre-divided in a set of sub-intervals with the equal length. This predetermined length reflects the degree of precision in efficiency measure fixed by the modeler. A counter is placed in each sub-interval to record the frequency of efficiency coefficient corresponding to this sub-interval. A histogram is obtained for each DMU and the corresponding efficiency distribution is determined by smoothing the histogram.

3.5. An Illustrative Example

To illustrate the methodology and for comparative purposes, consider the data contained in Cooper et al. (2001a) suumarized in *Table 1*. Details about the description of the real-world case, an example involving efficiency evaluations of the branch offices of a mobile telecommunications corporation in Korea, can be found in the cited reference.

Table 1: Imprecise Data for an Illustrative Example
(Adapted from Cooper et al., 2001a)

	X_1	X_2	X_3	Y_1	Y_2	Y_3
DMU 1	124	18.22	4	25.53	89.8	[80;85]
DMU 2	95	9.23	2	18.43	99.6	[85;90]
DMU 3	92	8.07	6	10.29	87	[75;80]
DMU 4	61	5.62	8	8.32	99.4	100
DMU 5	63	5.33	7	7.04	96.4	[70;75]
DMU 6	50	3.53	3	6.42	86	[90;95]
DMU 7	40	3.5	5	2.2	71	[80;85]
DMU 8	16	1.17	1	2.87	98	[95;100]

The GA based heuristics is used to determine the bounds for the efficiency confidence interval. The results are presented in *Table* 9. This table also contains the efficiency measures obtained by Cooper et al (2001a).

Table 2: Comparative results of *IDEA* and Simulated *Confident-DEA* with Imprecise Data

	DMU 1	**DMU 2**	**DMU 3**	**DMU 4**	**DMU 5**	**DMU 6**	**DMU 7**	**DMU 8**
Results of IDEA	1	1	0.894	1	0.976	1	0.895	1
Highest Efficiency	1	1	0.8363	0.8723	0.8746	0.9842	0.8113	1
Lowest Efficiency	1	1	0.5941	0.7159	0.2018	0.7414	0.2546	1

One can notice a small deviation from the exact optimal solution as determined by *IDEA*. This is due to the assumption of normality, which gives small weights to extreme values where the highest and lowest values of efficiency coefficients are most likely to be reached.

We define here a new concept: the *efficiency dominance*. This dominance can have three types:

(i) *First order efficiency dominance:* corresponds to a situation where the lowest value in the efficiency confidence interval of DMUi is greater than the highest value in the efficiency confidence interval of DMUj. In this situation, DMUi is strongly outperforming DMUj. We call this situation *First order efficiency dominance.* This is illustrated by the couple DMU2 and DMU4.

(ii) *Second order efficiency dominance:* corresponds to a situation where the two efficiency confidence interval of DMUi and DMUj are overlapping. In this situation, DMUi is weakly outperforming DMUj. We call this situation *second order efficiency dominance.* This is illustrated by the couple DMU5 and DMU6. Studying the distribution over each confidence interval will provide additional information about comparative performances. An interesting situation here is when an interval is totally included in the second. The DMU with narrower efficiency confidence interval range is likely to be considered outperforming since it has les volatile efficiency level. This is illustrated by the couple DMU5 and DMU7.

An interesting way of presenting the results is what we call the *Efficiency Cartogram*, presented in *Figure 4*.

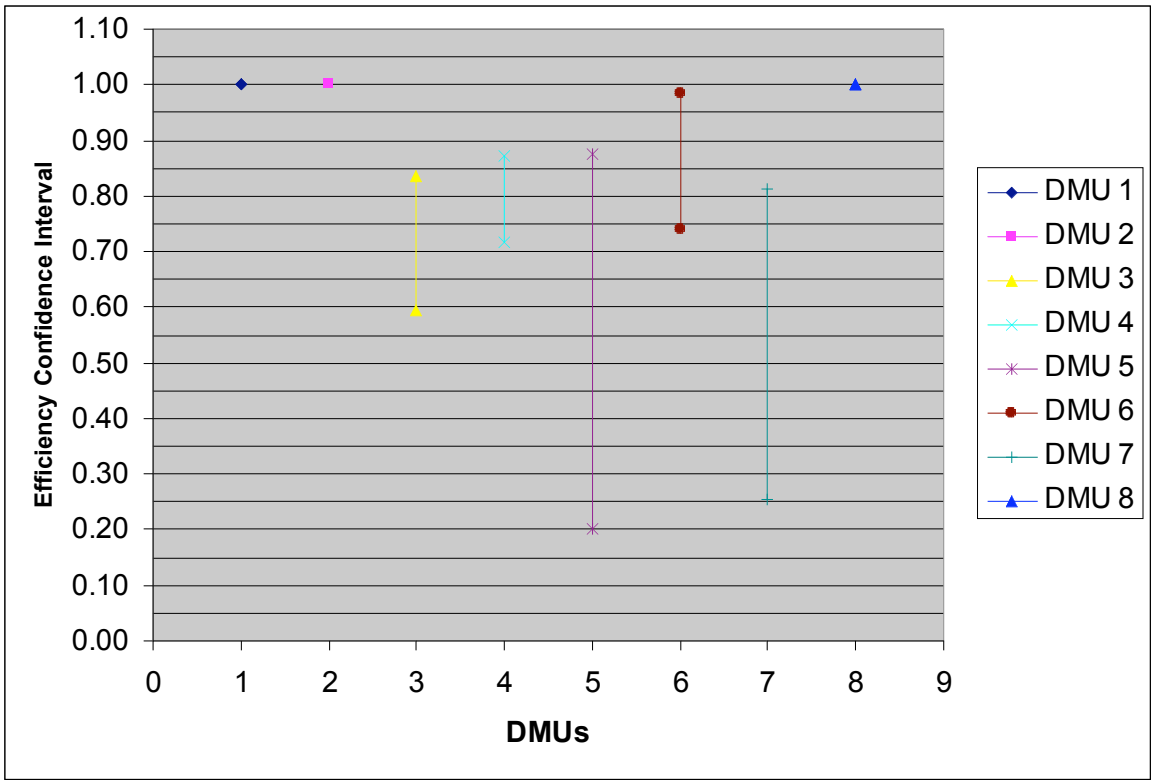


Figure 5: Efficiency Cartogram

To determine the distribution of the efficiency measures over their corresponding efficiency confidence interval, the simulation component of Confident-DEA was run 100,000 times. The results are provided in the appendix. The efficiency histogram for DMU6 is in *Figure 5*.

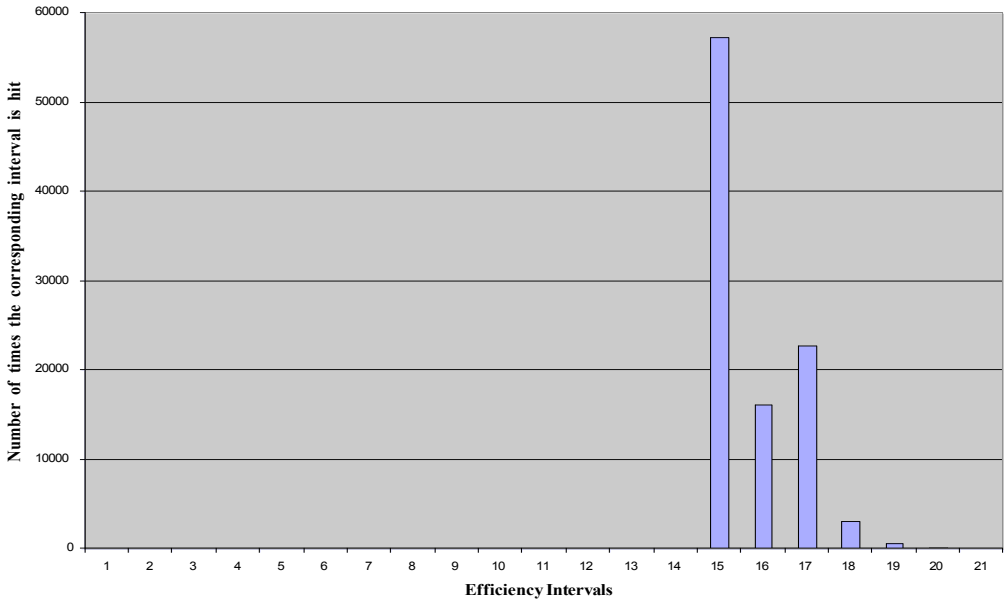


Figure 5: Efficiency Histogram for DMU 6

As in the case of cardinal bounded data, the Monte Carlo simulation component for *Confident-DEA* permits in the case of imprecise data the approximation of the distribution of efficiency values over the efficiency confidence interval.

4. Conclusions

Over its lifespan DEA had had its roots solidly planted in real world soils. I has enjoyed practitioner *acceptance*. As reported in Gattoufi et al. (2004a and 2004b), its *vitality* is demonstrated by the high rate of literature growth and by the large number of journals included. The diversity of journals having DEA content speaks to its *diffusion* and to its *relevance*. Real world situations often dictate data, the values of which lie within some prescribed bounds. Moreover, the data may be ordinal rather than cardinal in form, and hence known only to be satisfying certain ordinal relations. A new approach for dealing with the imprecision of data in DEA is presented in this paper, to further take into account the real world facts in the DEA analysis. Called *Confident-DEA*, this approach make it possible to reflect the imprecision in data in the final efficiency coefficients by providing an "efficiency confidence interval", hence the name *confident-DEA*, for those coefficients. This generalizes and improves the more traditional *IDEA* approach suggested first by Cooper et al. (1999). The spread of the efficiency confidence interval in any application may be considered as a measure of the "risk" attached to the corresponding DMU: the larger is the spread of the interval, the higher is the uncertainty in the level of the corresponding DMU's efficiency and therefore the higher is the risk attached to the corresponding DMU. Also, the spread can be an indicator of *volatility for the efficiency*. The wider is the spread, the more volatile is the efficiency and hence the less is the stability of the corresponding DMU in terms of efficiency.

This paper presents an original formulation of the general case of DEA with imprecise data, namely single valued and bounded data as well as ordinal data, in the form of a bi-level convex model. This NP-hard problem has no exact solving method in the literature. A genetic algorithm based solving method is suggested and represents an additional original contribution of the paper. The solving method proceeds in two steps. First each DMU is split up into a set of chromosomes, each one representing a *virtual single-valued* alternative for the real imprecise DMU. An *individual* is defined by a set of chromosomes determined by choosing, taking into account the distribution of imprecise values to make the approach stochastic, a representative from each set of virtual DMUs with exact data representing a real DMU. It is important to remark that, unlike the standard Genetic Algorithm procedure, an individual here is represented by a binary matrix rather then a binary string. The second step is to do the genetic modifications on the individuals. An originality in the genetic algorithm approach proposed here is to consider horizontal and vertical cutting for the crossover, unlike the traditional multiple cutting on the chromosome string commonly used in genetic algorithm. This is believed to improve the efficiency of the algorithm, although it needs to be proven.

Once the range for the efficiency was determined, a Monte-Carlo simulation based method was suggested to determine the distribution of the efficiency coefficients over the confidence interval. Significantly, IDEA always results in a single valued efficiency measure and implicitly assumes a uniform distribution for the bounded data. *Confident-DEA* on the other hand allows the use of any distribution for the bounded data. Additionally, the simulation component proposes benchmarks, in terms of inputs and outputs, for any DMU considered and for any desired level of efficiency included in its confidence interval. The use of simulation was dictated by the inexistence of analytical results about the relation between the data distribution(s) over their intervals and the distribution of the efficiency coefficients over the efficiency intervals. This remains an open research topic.

It can be affirmed that *Confident-DEA* generalizes *IDEA* in the sense that the efficiency levels identified by *IDEA* for each DMU coincide with the *optimistic point of view* in the *Confident-DEA* approach, in the case of bounded cardinal data, as introduced in Gattoufi (2002 and 2004).

Finally, like any heuristic method, the genetic algorithm solving method provides, one should remind, a satisfactory solution rather than an exact solution. Hence, other solving method can be suggested and compared with the results provided by the solving method described and illustrated in this work.

A potential application for this is analysing the performance of shares in a stock market and providing advices about their future performances. Such work, in the best of our knowledge will represent an originality in the applications of DEA in general and Confident-DEA in particular. Another potential application of Confident-DEA is in predicting efficiency. Given the relative nature of the efficiency measures, forecasting efficiency measures cannot be done directly by using a time-series methods or econometric modelling. By predicting the production factors, one can generate prediction confidence intervals. These intervals are then considered as imprecise bounded data in a *Confident-DEA* approach to provide the efficiency confidence interval for each DMU. The results obtained from the simulation component can be used to define a parametric approximation for the distribution of efficiency measures over their corresponding confidence intervals.

References:

Bard, J. F. (1998), *Practical Bilevel Optimization: Algorithms and Applications*, Kluwer Academic Publishers, Dordrecht, The Netherlands.

Charnes, A., Cooper, W.W. (1985), "Preface to topics in Data Envelopment Analysis", Annals of Operations Research 2, 1, 59-94.

Charnes, A., Cooper, W.W., and Rhodes, E. (1978), "Measuring the efficiency of decision making units" European Journal of Operational Research 2, 429-444.

Cook, W. D., Kress, M. and Seiford, L. M. (1993), "On the use of ordinal data in Data Envelopment Analysis", Journal of Operational Research Society 44, 2, 133-140.

Cook, W. D., Kress, M. and Seiford, L. M. (1996), "Data Envelopment Analysis in the presence of both quantitative and qualitative Factors", Journal of Operational Research Society 47, 7, 945-953.

Cooper, W.W., Park, K. S. and Yu, G. (2001a), "An illustrative application of IDEA (Imprecise Data Envelopment Analysis) to a Korean Mobile Telecommunication Company", Operations Research 49, 6, 807-820.

Cooper, W.W., Park, K. S. and Yu, G. (2001b), "IDEA (Imprecise Data Envelopment Analysis) with CMDS (Column Maximum Decision Making Units)", Journal of Operational Research Society, 52, 2, 178-181

Cooper, W. W., Park, K. S. and Yu, G. (1999), "IDEA and AR-IDEA: models for dealing with imprecise data in DEA", Management Science 45, 4, 597-607.

Gattoufi, S. 2002. Data Envelopment Analysis: A taxonomy, A meta review and an extension (Confident-DEA) with application to predicting cross (OECD) country banking systems' efficiency. Ph.D. Dissertation, Graduate School of Management, Sabanci University, Istanbul, Turkey.

Gattoufi, S., Reisman, A., 2004. Confident-DEA: A new methodology for efficiency analysis with imprecise data. In Modeling, Computation and Optimization in Information Systems and Management Sciences edited by Le Thi Hoai An, *and* Pham Dinh Tao

Gattoufi, S., Oral, M., Kumar, A., Reisman, A., 2004. Content Analysis of Data Envelopment Analysis Literature and its Comparison with that of other OR/MS Fields. *Journal of the Operational Research Society* 55(9), 911-932.

Gattoufi, S., Oral, M., Kumar, A., Reisman, A., 2004. Epistemology of Data Envelopment Analysis and Comparison with Other Fields of OR/MS for Relevance to Applications. *Socio Economic Planning Sciences* 38(2-3), 123-140.

Golany, B. (1988), "A note on including ordinal relations among multipliers in Data Envelopment Analysis", Management Science 34, 8, 1029-1033.

Goldberg, D. E. (1989), *Genetic algorithms in search, optimization and machine learning*, Addison-Wesley Publishing Company, Reading, Massachusetts.

Hansen, P., Jaumard, B. and Savard, G. (1992), "New Branch-and-Bound rules for linear bilevel programming", SİAM Journal of Scientific and Statistical Computing, 13, 5, 1194-1217.

Holland, J. H. (1992), *Adaptation in Natural and Artificial Systems*, Second Edition, The MIT Press, Cambridge, Massachusetts.

Jeroslow, R. G. (1985), "The polynomial hierarchy and a simple model for competitive analysis", Mathematical Programming, 32, 146-164.

Kim, H. S., Park, C. G. and Park, K. S. (1999), "An application of Data Envelopment Analysis in telephone offices evaluation with partial data", Computers & Operations Research 26, 1, 59-72.

Lee**,** Y**.**K**.**, Park**,** K**.**S**.**, Kim**,** S**.**H**.** (2002), "Identification of inefficiencies in an additive model based IDEA (imprecise data envelopment analysis)". Computers & Operations Research 29, 12 ,1661-76.

Man, K. F., Tang, K. S., Kwong, S. **(**1999**),** *Genetic algorithms: concepts and designs*. Springer-Verlag, London, England.

Post, T., Spronk, J., (1999), "Performance benchmarking using interactive data envelopment analysis", European Journal of Operational Research, 115, 3, 472-487.

Author Index