The 18th European Conference on Artificial Intelligence

Proceedings

Workshop on Modeling and Solving Problems with Constraints

Monday July 21, 2008
Patras, Greece

Steve Prestwich, Claude-Guy Quimper, and Toby Walsh

Preface

The Workshop on Modeling and Solving Problems with Constraints was held in Patras (Greece) on July the 21rst 2008 jointly with the 18th European Conference on Artificial Intelligence (ECAI 2008). The objective of this workshop is to promote discussion of novel ideas related to constraint programming. The program committee selected 9 papers from the submissions to appear in these proceedings and to be presented at the workshop. We would like to thank the program committee for reviewing the papers and sharing their comments with the authors. We would also like to thank the external reviewers.

July 2008

Steve Prestwich
Claude-Guy Quimper
Toby Walsh

Organizers

Steve Prestwich
Claude-Guy Quimper
Toby Walsh

Program committee

Peter van Beek, University of Waterloo, Canada
Nicolas Beldiceanu, Ecole des mines de Nantes, France
Brahim Hnich, Izmir University of Economics, Turkey
Ulrich Junker, ILOG, France
Javier Larrosa, Universitat Politecnica de Catalunya, Spain
Ian Miguel, University of St Andrews, United Kingdom
Gilles Pesant, Ecole polytechnique de Montreal, Canada
Steve Prestwich, University College Cork, Ireland
Claude-Guy Quimper, Ecole polytechnique de Montréal, Canada
Toby Walsh, NICTA and UNSW, Australia

External Reviewers

Louis-Martin Rousseau
Peter Nightingale

Table of Content

Exploiting Constraint Weights for Revision Ordering in Arc Consistency Algorithms

Thanasis Balafoutis and **Kostas Stergiou**
Department of Information & Communication Systems Engineering
University of the Aegean, Greece
email: {abalafoutis,konsterg}@aegean.gr

Abstract. Coarse grained arc consistency algorithms, like AC-3, operate by maintaining a list of arcs (or variables) that records the revisions that are still to be performed. It is well known that the performance of such algorithms is affected by the order in which revisions are carried out. As a result, several heuristics for ordering the elements of the revision list have been proposed. These heuristics exploit information about the original and the current state of the problem, such as domain sizes, variable degrees, and allowed combinations of values, to reduce the number of constraint checks and list operations aiming at speeding up arc consistency computation. Recently, Boussemart et al. proposed novel variable ordering heuristics that exploit information about failures gathered throughout search and recorded in the form of constraint weights. Such heuristics are now considered as the most efficient general purpose variable ordering heuristic for CSPs. In this paper we show how information about constraint weights can be exploited to efficiently order the revision list when AC is applied during search. We propose a number of simple revision ordering heuristics based on constraint weights for arc, variable, and constraint oriented implementations of coarse grained arc consistency algorithms, and compare them to the most efficient existing revision ordering heuristic. Importantly, the new heuristics can not only reduce the numbers of constraints checks and list operations, but also cut down the size of the explored search tree. Results from various structured and random problems demonstrate that some of the proposed heuristics can offer significant speed-ups.

1 Introduction

Among the plethora of algorithms that have been devised to solve CSPs, the look-ahead algorithm termed MAC (maintaining arc consistency) [16, 2] is considered as one of the most efficient. As MAC applies arc consistency (AC) on the problem after every variable assignment, speeding up the process of AC application has received a lot of attention in the literature. The numerous AC algorithms that have been proposed can be classified into *coarse grained* and *fine grained*. Typically, coarse grained algorithms like AC-3 [12] and its extensions (e.g. AC2001/3.1 [3] and AC-3_d [7]) apply successive revisions of arcs or, depending on the implementation, variables [13]. On the other hand, fine grained algorithms like AC-2004 [14] and AC-2007 [1] use various data structures to apply successive revisions of variable-value-constraint triplets. Although AC-3 does not have an optimal worst-case time complexity, as the fine grained algorithms do, it is competitive and often better in practice and has the additional advantage of being easy to implement. Further to this, some of the extensions to AC-3 achieve optimal worst-case complexity while preserving the simplicity of implementation and good average case behavior.

It is well known that the way in which the list of revisions is implemented and manipulated is an important point regarding the efficiency of coarse grained AC algorithms. In a recent empirical investigation Boussemart et al. [4] showed that a variable-oriented implementation of AC-3, as proposed in [13], usually outperforms the standard arc-oriented implementation of [12] and the constraint-oriented implementation of [4]. Perhaps more significantly, the order in which the elements of the revision list are processed, in any implementation, can have a notable effect on the number of constraint checks and list insertion/removal operations. Since MAC applies AC thousands or even millions of times during search, any savings in checks and list operations can be reflected on the overall cpu time efficiency. Having recognized this, Wallace and Freuder proposed a number of revision ordering heuristics aiming at speeding up AC processing as early as 1992. Since then this issue has been further investigated and alternative heuristics have been proposed [8, 7, 11, 4]. All the proposed heuristics exploit information about the original and the current state of the problem, such as domain sizes, variable degrees, and allowed combinations of values, to reduce the number of constraint checks and list operations. However, it has to be noted that even the most successful revision heuristic of variable-oriented propagation only offers a 25% speed-up compared to a *fifo* implementation of the revision list [4].

In recent years, powerful variable ordering heuristics have been proposed and their integration with MAC has led to significant speed-ups of existing solvers. The *conflict-driven* weighted degree (*wdeg*) heuristics of Boussemart et al. are designed to enhance variable selection by incorporating knowledge gained during search, in particular knowledge derived from failures [5]. These heuristics work as follows. All constraints are given an initial weight of 1. During search the weight of a constraint is incremented by 1 every time the constraint causes a domain wipe-out (DWO) during constraint

propagation. The weighted degree (*wdeg*) of a variable is then the sum of the weights of the constraints that include this variable and at least another unassigned variable. The weights are continuously updated during search by using information learnt from previous failures. The basic *wdeg* heuristic selects the variable having the largest weighted degree. In addition to the basic *wdeg* heuristic, combining weighted degree and domain size yields a heuristic that selects the variable with the smallest ratio of current domain size to current weighted degree (*dom/wdeg*). The advantage that these heuristics offer is that they use previous search states as guidance, while older standard heuristics either use the initial state or the current state only.

In this paper we show how information about constraint weights can be exploited not only to perform variable selection, but also to efficiently order the revision list when AC is maintained during search. We investigate several new constraint weight based approaches to ordering the revision list in all the alternative implementations of AC-3: arc-oriented, variable-oriented and constraint-oriented. Experimental results from various random, academic and real world problems show that some of the proposed heuristics, when used in conjunction with a conflict-driven variable ordering heuristic such as dom/wdeg, demonstrate a measurable improvement in constraint checks compared to the most efficient existing revision ordering heuristic.

Notably, the new revision heuristics can not only reduce the numbers of constraint checks and list operations, but also cut down the size of the explored search tree by focusing search on more relevant variables. Due to this, in the variable-oriented implementation of AC-3, which is the most efficient among the three alternatives, the new heuristics can offer significant savings in cpu times. This opens up interesting directions for future work since, apart from an implementation tool, revision ordering heuristics can be viewed as methods to have a really important impact on the search process.

The rest of the paper is organized as follows. Section 2 gives the necessary definitions and notation and briefly describes the three alternative implementations of AC-3. Section 3 summarizes existing work on revision ordering heuristics for constraint propagation. In Section 4 we propose new revision ordering heuristics based on constraint weights. In Section 5 we experimentally compare the proposed heuristics to the best existing revision heuristics on a variety of problems. Conclusions are presented in Section 6.

2 Background

A *Constraint Satisfaction Problem* (CSP) is a tuple (*X, D, C*), where X is a set containing n variables $\{x_1, x_2, ..., x_n\}$; D is a set of domains $\{D(x_1), D(x_2), ..., D(x_n)\}$ for those variables, with each $D(x_i)$ consisting of the possible values which x_i may take; and C is a set of constraints $\{c_1, c_2, ..., c_k\}$ between variables in subsets of X. Each $c_i \in C$ expresses a relation defining which variable assignment combinations are allowed for the variables in the scope of the constraint, *vars*(c_i). Two variables are said to be *neighbors* if they share a constraint. The *arity* of a constraint is the number of variables in the scope of the constraint. A binary constraint between variables x_i and x_j will be denoted by c_{ij}. In this paper we focus on binary CSPs. However, the proposed revision ordering heuristics are generic and can be applied on problems with constraints of any arity.

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

An *arc* is a pair (c, x_i) where $x_i \in vars(c)$. As we focus on binary CSPs, any arc (c_{ij}, x_i) will be alternatively denoted by the pair of variables (x_i,x_j), where $x_j \in vars(c_{ij})$. That is, x_j is the other variable involved in c_{ij}. An arc (x_i,x_j) is *arc consistent* (AC) iff for every value $a \in D(x_i)$ there exists at least one value $b \in D(x_j)$ such that the pair (a,b) satisfies c_{ij}. In this case we say that b is a *support* of a on arc (x_i,x_j). Accordingly, a is a support of b on arc (x_j,x_i). A problem is AC iff there are no empty domains and all arcs are AC. The application of AC on a problem results in the removal of all non-supported values from the domains of the variables. A *support check* (consistency check) is a test to find out if two values support each other. The *revision* of an arc (x_i,x_j) using AC verifies if all values in $D(x_i)$ have supports in $D(x_j)$. We say that a revision is *fruitful* if it deletes at least one value, while it is *redundant* if it achieves no pruning. A *DWO-revision* is one that causes a *DWO*. That is, it results in an empty domain.

In the following, we will use the basic coarse grained algorithm to establish arc consistency, namely, AC-3. This does not limit the generality of the proposed heuristics as they can be easily integrated into any coarse grained AC algorithm. In the reported experiments we use MAC as our search algorithm and the *dom/wdeg* heuristic for dynamic variable ordering.

2.1 AC-3 variants

The AC-3 arc consistency algorithm can be implemented using a variety of propagation schemes. We recall here the three variants, as presented in [4], which respectively correspond to algorithms with an arc-oriented, variable-oriented and constraint-oriented propagation scheme.

The first one (arc-oriented propagation) is the most commonly presented and used because of its simple and natural structure. Algorithm 1 depicts the main procedure. As explained, an arc is a variable pair (x_i, x_j) which corresponds to a directed constraint. Hence, for each binary constraint c_{ij} involving variables x_i and x_j there are two arcs, (x_i, x_j) and (x_j, x_i). Initially, the algorithm inserts all arcs in the revision list Q. Then, each arc (x_i, x_j) is removed from the list and revised in turn. If any value in $D(x_i)$ is removed when revising (x_i, x_j), all arcs pointing to x_i (i.e. having x_i as second element in the pair), except (x_i, x_j), will be inserted in Q (if not already there) to be revised. Algorithm 2 depicts function *revise*(x_i, x_j) which seeks supports for the values of x_i in $D(x_j)$. It removes those values in $D(x_i)$ that do not have any support in $D(x_j)$. The algorithm terminates when the list Q becomes empty.

The variable-oriented propagation scheme was proposed by McGregor [13] and later studied in [6]. Instead of keeping arcs in the revision list, this variant of AC-3 keeps variables. The main procedure is depicted in Algorithm 3. Initially, all variables are inserted in the revision list Q. Then each variable x_i is removed from the list and each constraint involving x_i

Algorithm 1 ARC-ORIENTED AC3

```
1: Q ←{(x_i, x_j)} | c_ij ∈ C or c_ji ∈ C, i ≠ j
2: while Q ≠ ∅ do
3:    select and delete an arc (x_i, x_j) from Q
4:    if REVISE(x_i, x_j) then
5:       Q ← Q ∪ {(x_k, x_i)} | c_ki ∈ C, k ≠ j
6:    end if
7: end while
```

Algorithm 2 REVISE-3(x_i, x_j)

```
1: DELETE ← false
2: for each a ∈ D(x_i) do
3:    if ∄ b ∈ D(x_j) such that (a, b) satisfies c_ij then
4:       delete a from D(x_i)
5:       DELETE ← true
6:    end if
7: end for
8: return DELETE
```

Algorithm 3 VARIABLE-ORIENTED AC3

```
1: Q ← {x_i | x_i ∈ X}
2: ∀ c_ij ∈ C, ∀x_i ∈ vars(c_ij), ctr(c_ij, x_i) ← 1
3: while Q ≠ ∅ do
4:    get x_i from Q
5:    for each c_ij | x_i ∈ vars(c_ij) do
6:       if ctr(c_ij, x_i) = 0 then continue
7:       for each x_j ∈ vars(c_ij) do
8:          if needsNotBeRevised(c_ij, x_j) then continue
9:          nbRemovals ← revise(c_ij, x_j)
10:         if nbRemovals > 0 then
11:            if dom(x_j) = ∅ then return false
12:            Q ← Q ∪ {x_j}
13:            for each c_jk | c_jk ≠ c_ij ∧ x_j ∈ vars(c_jk) do
14:               ctr(c_jk, x_j) ← ctr(c_jk, x_j) + nbRemovals
15:            end for
16:         end if
17:      end for
18:      for each x_j ∈ vars(c_ij) do ctr(c_ij, x_j) ← 0
19:   end for
20: end while
21: return true
```

Algorithm 4 needsNotBeRevised(c_{ij}, x_i) : boolean

```
1: return (ctr(c_ij, x_i) > 0 and ∄x_j ∈ vars(c_ij) | x_j ≠ x_i ∧
   ctr(c_ij, x_j) > 0)
```

is processed. For each such constraint c_{ij} we revise the arc (x_j,x_i). If the revision removes some values from the domain of x_j, then variable x_j is inserted in Q (if not already there).

Function *needsNotBeRevised* given in Algorithm 4, is used to determine relevant revisions. This is done by associating a counter $ctr(c_{ij},x_i)$ with any arc (x_i,x_j). The value of the counter denotes the number of removed values in the domain of variable x_i since the last revision involving constraint c_{ij}. If x_i is the only variable in $vars(c_{ij})$ that has a counter value greater than zero, then we only need to revise arc (x_j,x_i). Otherwise, both arcs are revised.

The constraint-oriented propagation scheme is depicted in Algorithm 5. This algorithm is an analogue to Algorithm 3.

Algorithm 5 CONSTRAINT-ORIENTED AC3

```
1: Q ← {c_ij | c_ij ∈ C}
2: ∀ c_ij ∈ C, ∀x_i ∈ vars(c_ij), ctr(c_ij, x_i) ← 1
3: while Q ≠ ∅ do
4:    get c_ij from Q
5:    for each x_j ∈ vars(c_ij) do
6:       if needsNotBeRevised(c_ij, x_j) then continue
7:       nbRemovals ← revise(c_ij, x_j)
8:       if nbRemovals > 0 then
9:          if dom(x_j) = ∅ then return false
10:         for each c_jk | c_jk ≠ c_ij ∧ x_j ∈ vars(c_jk) do
11:            Q ← Q ∪ {x_j}
12:            ctr(c_jk, x_j) ← ctr(c_jk, x_j) + nbRemovals
13:         end for
14:      end if
15:   end for
16:   for each x_j ∈ vars(c_ij) do ctr(c_ij, x_j) ← 0
17: end while
18: return true
```

Initially, all constraints are inserted in the revision list Q. Then each constraint c_{ij} is removed from the list and each variable $x_j \in vars(c_{ij})$ is selected and revised. If the revision of the selected arc (c_{ij}, x_j) is fruitful, then the reinsertion of the constraint c_{ij} in the list is needed. As in variable-oriented scheme, the same counters are also used here to avoid useless revisions.

3 Related work

Revision ordering heuristics is a topic that has received considerable attention in the literature. The first systematic study on this topic was carried out by Wallace and Freuder, who proposed a number of different revision ordering heuristics that can be used with the arc-oriented variant of AC3 [17]. These heuristics, which are defined for binary constraints, are based on three major features of CSPs: (i) the number of acceptable pairs in each constraint (the constraint size or satisfiability), (ii) the number of values in each domain and (iii) the number of binary constraints that each variable participates in (the degree of the variable). Based on these features, they proposed three revision ordering heuristics: (i) ordering the list of arcs by increasing relative satisfiability (*sat up*), (ii) ordering by increasing size of the domain of the variables (*dom j up*) and (iii) ordering by descending degree of each variable (*deg down*).

The heuristic *sat up* counts the number of acceptable pairs of values in each constraint (i.e the number of tuples in the Cartesian product built from the current domains of the variables involved in the constraint) and puts constraints in the list in ascending order of this count. Although this heuristic reduces the list additions and constraint checks, it does not speed up the search process. When a value is deleted from the domain of a variable, the counter that keeps the number of acceptable arcs has to be updated. This process is usually time consuming because the algorithm has to identify the constraints in which the specific variable participates and to recalculate the counters with acceptable value pairs. Also an additional overhead is needed to reorder the list.

The heuristic *dom j up* counts the number of remaining values in each variable's current domain during search. Vari-

ables are inserted in the list by increasing size of their domains. This heuristic reduces significantly list additions and constraint checks and is the most efficient heuristic among those proposed in [17].

The *deg down* heuristic counts the current degree of each variable. The initial *degree* of a variable x_i is the number of variables that share a constraint with x_i. During search, the *current degree* of x_i is the number of unassigned variables that share a constraint with x_i. The *deg down* heuristic sorts variables in the list by decreasing size of their current degree. As noticed in [17] and confirmed in [4], the (*deg down*) heuristic does not offer any improvement.

Gent et al. [8] proposed another heuristic called k_{ac}. This heuristic is based on the number of acceptable pairs of values in each constraint and tries to minimize the constrainedness of the resulting subproblem. Experiments have shown that k_{ac} is time expensive but it performs less constraint checks when compared to *sat up* and *dom j up*.

Boussemart et al. performed an empirical investigation of the heuristics of [17] with respect to the different variants (arc, variable and constraint) of AC-3 [4]. In addition, they introduced some new heuristics. Concerning the arc-oriented AC-3 variant, they have examined the *dom j up* as a stand alone heuristic (called dom^v) or together with *deg down* which is used in order to break ties (called $ddeg \circ dom^v$). Moreover, they proposed the ratio *sat up/dom j up* (called dom^c/dom^v) as a new heuristic. Regarding the variable-oriented variant, they adopted the dom^v and *ddeg* heuristics from [17] and proposed a new one called rem^v. This heuristic corresponds to the greatest proportion of removed values in a variable's domain. For the constraint-oriented variant they used dom^c (the smallest current domain size) and rem^c (the greatest proportion of removed values in a variable's domain). Experimental results showed that the variable-oriented AC-3 implementation with the dom^v revision ordering heuristic (simply denoted *dom* hereafter) is the most efficient alternative.

4 Revision ordering heuristics based on constraint weights

The heuristics described in the previous section, and especially *dom*, improve the performance of AC-3 (and MAC) when compared to the classical queue or stack implementation of the revision list. This improvement in performance is mainly due to the reduction in list additions and constraint checks. A key principle that can also have a positive effect on the performance is the "fail-first principle" of Haralick and Elliot [10] which states that "to succeed, try first where you are most likely to fail". Considering revision ordering heuristics this principle can be translated as follows: When AC is applied during search (within an algorithm such as MAC), to reach as early as possible a failure (*DWO*), order the revision list by putting first the arc or variable which will guide you earlier to a *DWO*.

To apply the "fail-first principle" in revision ordering heuristics, we must use some metric to compute which arc (or variable) in the AC revision list is the most probable to cause failure. Until now, constraint weights have only been used for variable selection. In our proposed revision ordering heuristics, we use information about constraint weights as a metric to order the AC revision list. These heuristics can efficiently be used in conjunction with conflict-driven variable ordering heuristics in order to boost search.

The main idea behind these new heuristics is to handle as early as possible potential *DWO-revisions* by appropriately ordering the arcs, variables, or constraints in the revision list. In this way the revision process of AC will be terminated earlier and thus constraint checks can be significantly be reduced. Moreover, with such a design we may be able to avoid many *redundant revisions*.

Revision ordering and variable ordering heuristics have different tasks to perform when used in a search algorithm like MAC. Before the appearance of conflict-driven heuristics there was no way to achieve an interaction with each other, i.e. the order in which the list was organized during AC was impossible to affect the decision of which variable to select next (and vice versa). The contribution of revision ordering heuristics to the solver's efficiency was limited to the reduction of list additions and constraint checks.

However, when a conflict-driven variable ordering heuristic like *wdeg* or *dom/weg* is used, then there are cases where the decision of which arc (or variable) to revise first can affect the variable selection. To better illustrate this interaction we give the following example.

Example 1 Assume we are using MAC with an arc-oriented implementation of AC-3 to solve a CSP (X, D, C). Also assume that a conflict-driven variable ordering heuristic (e.g. dom/wdeg) is used, and that at some point during search the following AC revision list is formed: Q=$\{(c_{12}, x_1), (c_{34}, x_3), (c_{56}, x_5)\}$. Suppose that (c_{12}, x_1) and (c_{56}, x_5) can both lead to a DWO if they are selected first from the list. If a revision ordering heuristic R_1 selects (c_{12}, x_1) first then the DWO of x_1 will be detected and the weight of constraint c_{12} will increased by 1. If some other revision ordering heuristic R_2 selects (c_{56}, x_5) first then the DWO of x_5 will be detected but this time the weight of a different constraint (c_{56}) will increased by 1. Since constraint weights affect the choices of the variable ordering heuristic, R_1 and R_2 can lead to different future decisions for variable instantiation. Thus, R_1 and R_2 may guide search to different parts of the search space.

We now describe a number of new revision ordering heuristics for all three AC-3 variants. It is easy to see that all these heuristics are lightweight (i.e. cheap to compute) assuming that the weights of constraints are updated during search.

Arc-oriented heuristics are tailored for the arc-oriented variant where the list of revisions Q stores arcs of the form (c_{ij},x_i). Since an arc consists of a constraint c_{ij} and a variable x_i, we can use information about the weight of the constraint, or the weight of the variable, or both, to guide the heuristic selection. These ideas are the basis of the proposed heuristics described below. For each heuristic we specify the arc that it selects.

- *wcon*: selects the arc (c_{ij},x_i) such that c_{ij} has the highest weight *wcon* among all constraints appearing in an arc in Q.
- *wdeg*: selects the arc (c_{ij},x_i) such that x_i has the highest weighted degree *wdeg* among all variables appearing in an arc in Q.
- *dom/wdeg*: selects the arc (c_{ij},x_i) such that x_i has the smallest ratio between current domain size and weighted degree among all variables appearing in an arc in Q.

- *dom/wcon*: selects the arc (c_{ij},x_i) having the smallest ratio between the current domain size of x_i and the weight of c_{ij} among all arcs in Q.

The call to one of the proposed arc-oriented heuristics can be attached to line 3 of Algorithm 1.

Variable-oriented heuristics are tailored for the variable-oriented variant of AC-3 where the list of revisions Q stores variables. For each of the heuristics given below we specify the variable that it selects.

- *wdeg*: selects the variable having the highest weighted degree *wdeg* among all variables in Q.
- *dom/wdeg*: selects the variable having the smallest ratio between current domain size and *wdeg* among all variables in Q.

The call to one of the proposed variable-oriented heuristics can be attached to line 4 of Algorithm 3. After selecting a variable, the algorithm revises, in some order, the constraints in which the selected variable participates (line 5). Our heuristics process these constraints in descending order according to their corresponding weight.

Finally, the constraint-oriented heuristic *wcon* selects a constraint c_{ij} from the AC revision list having the highest weight among all constraints in Q. The call to this heuristic can be attached to line 4 of Algorithm 5. One can devise more complex constraint-oriented heuristics by aggregating the weighted degrees of the variables involved in a constraint. However, we have not yet experimented with such heuristics.

5 Experiments and results

In this section we experimentally investigate the behavior of the new revision ordering heuristics proposed above on several classes of real, toy and random problems[1]. In our experiments we included both satisfiable and unsatisfiable instances. We only give results for the two most significant arc consistency variants: arc and variable oriented. We have excluded the constraint-oriented variant since this is not as competitive as the other two [4].

We compare our heuristics with *dom*, the most efficient previously proposed revision ordering heuristic. We also include results from the standard *fifo* implementation of the revision list which always selects the oldest element in the list (i.e. the list is implemented as a queue). In our tests we have used the following measures of performance: cpu time in seconds (t), number of visited nodes (n) and number of constraint checks (c). The solver we used applies d-way branching, dom/wdeg for variable ordering and lexicographic value ordering. It also employs restarts. Concerning the restart policy, the initial number of allowed backtracks for the first run has been set to 10 and at each new run the number of allowed backtracks increases by a factor of 1.5.

Tables 1 and 2 show results from some real-world RLFAP instances. In the arc-oriented implementation of AC-3 (Table 1), heuristics *wcon*, mainly, and *dom/wcon*, to a lesser extent, decrease the number of constraint checks compared to *dom*. However, the decrease is not substantial and is rarely translated into a decrease in cpu times. The notable speed-up observed for problem s11-f6 is mainly attributed to the reduction in node visits offered by the two new heuristics. *wdeg* and *dom/wdeg* are less competitive, indicating that information about the variables involved in arcs is less important compared to information about constraints.

[1] (http://www.cril.univ-artois.fr/∼lecoutre/research/benchmarks/)

Table 1. Cpu times (t), constraint checks (c) and nodes (n) from frequency allocation problems (hard instances) using arc and variable oriented propagation. The s prefix stands for scen instances. Best cpu time is in bold.

		ARC ORIENTED					
Inst.		*queue*	*dom*	*wcon*	*wdeg*	*d/wdeg*	*d/wcon*
s11-f9	t	17,1	**11,7**	13,3	13,5	17,3	12,9
	c	18M	13,9M	9,5M	15M	15,1M	12,1M
	n	1760	1688	1689	1671	1681	1697
s11-f8	t	34,2	**18,5**	20,5	20	26	21,4
	c	33,5M	21,1M	13,8M	21,7M	23,7M	19,8M
	n	2902	2679	2699	2746	2682	2822
s11-f7	t	234,6	**133,5**	154,9	241,7	187,5	297,2
	c	193,1M	114,7M	92,5M	202,5M	147,6M	215,9M
	n	25830	21571	23334	30185	22427	43695
s11-f6	t	518	423,9	**281,9**	492,4	760,8	361,2
	c	347M	336,9M	166,1M	372,1M	536,3M	261M
	n	68225	73235	42541	71918	99874	52512
s11-f5	t	2571	2102	2792	2947	2641	**2088**
	c	1,793G	1,539G	1,509G	2,107G	1,868G	1,414G
	n	310,4M	318,3M	440,2M	378,1M	272,3M	274,3M
s11-f4	t	10220	**7084**	7523	9464	11409	9543
	c	7,150G	5,075G	3,812G	6,490G	7,706G	6,186G
	n	1,103G	1,038G	1,116G	1,219G	1,245G	1,152G

The variable-oriented implementation (Table 2) is clearly more efficient than the arc-oriented one. This confirms the results of [4]. Concerning this implementation, heuristic *dom/wdeg* outperforms *dom* and *queue* both in node visits and checks. Importantly, these savings are reflected on notable cpu time gains making the variable-oriented *dom/wdeg* the overall winner. Results also show that as the instances becomes harder, the efficiency of *dom/wdeg* heuristic compared to *dom* increases. The variable-oriented *wdeg* heuristic in most cases outperforms *dom* but is clearly less efficient than *dom/wdeg*.

Table 2. Cpu times (t), constraint checks (c) and nodes (n) from frequency allocation problems (hard instances) using arc and variable oriented propagation. The s prefix stands for scen instances. Best cpu time is in bold.

		VARIABLE ORIENTED			
Inst.		*queue*	*dom*	*wdeg*	*d/wdeg*
s11-f9	t	16,2	9,3	9,9	**9**
	c	16,3M	8,2M	9,3M	7,9M
	n	1767	1635	1677	1664
s11-f8	t	30,1	15,8	16,9	**15,2**
	c	30,3M	12,4M	14,7M	12,1M
	n	2879	2697	2679	2695
s11-f7	t	187,3	144,1	140,8	**98,6**
	c	139,1M	84,1M	113,4M	59,5M
	n	26139	27485	21332	19298
s11-f6	t	286	356,3	395,8	**245,6**
	c	220,3M	189,4M	297,6M	138,6M
	n	36331	68391	60919	46174
s11-f5	t	2254	2966	1840	**1579**
	c	1,492G	1,522G	1,081G	832,8M
	n	327,9M	582,1M	278,9M	292,6M
s11-f4	t	12729	10806	8648	**6077**
	c	8,676G	5,405G	4,975G	3,110G
	n	1,682G	1,982G	1,374G	1,048G

In Table 3 we present results from structured instances belonging to benchmark classes *langford* and *driver*. As the variable-oriented AC-3 variant is more efficient than the arc-oriented one, we only present results from the former. Results show that on easy problems all heuristics except *queue* are quite competitive. But as the difficulty of the problem increases, the improvement offered by the *dom/wdeg* revision heuristic becomes clear. On instance driverlogw-09 we can see

the effect that weight based revision ordering heuristics can have on search. *dom/wdeg* cuts down the number of node visits by more than 5 times resulting in a similar speed-up. It is interesting that *dom/wdeg* is considerably more efficient than *wdeg* and *dom*, indicating that information about domain size or weighted degree alone is not sufficient to efficiently order the revision list.

Table 3. Cpu times (t), constraint checks (c) and nodes (n) from structured problems using variable oriented propagation. Best cpu time is in bold.

Instance		*queue*	*dom*	*wdeg*	*d/wdeg*
langford-2-9	t	49,4	42,7	55	**42,1**
	c	71,7M	58,8M	71,9M	58,7M
	n	71729	58897	71907	59095
langford-2-10	t	450	392,4	381,7	**309,9**
	c	241,8M	204,1M	198M	142,4M
	n	497,359	410819	381161	305480
langford-3-11	t	633,5	590,9	768,6	**467,9**
	c	294M	253,8M	337,3M	184,7M
	n	119036	99619	152063	96567
langford-4-10	t	76,3	52,6	90,6	**37,5**
	c	37,8M	23,9M	42,9M	15,6M
	n	5253	4352	5759	3896
driverlogw-08c	t	26,4	13,4	**13,1**	13,3
	c	15M	6,2M	7,9M	6,5M
	n	7576	4451	2870	3895
driverlogw-09	t	206,1	374,5	315,6	**63,9**
	c	109M	181M	146,5M	28,4M
	n	30720	60084	46188	10917

Finally, in Table 4 we present results from benchmark random problems. Here, there is a large diversity in the results. All heuristics seems to lack robustness and there is no clear winner. The constraint weight based heuristics can be up to one order of magnitude faster than *dom* (instance geo50-20-d4-75-2), but they can also be significantly slower (frb30-15-2). In all cases, the large run time differences in favor of one or another heuristic are caused by corresponding differences in the size of the explored search tree, as node visits clearly demonstrate.

A possible explanation for the diversity in the performance of the heuristics on random problems as opposed to structured ones is the following. When dealing with structured problems, and assuming we use the variable-oriented variant of AC-3, a weighted based heuristic like *dom/wdeg* will give priority for revision to variables that are involved in hard subproblems and hence will carry out DWO-revisions faster. This will in turn increase the weights of constraints that are involved in such hard subproblems and thus search will focus on the most important parts of the search space. Random instances that lack structure do not in general consist of hard local subproblems. Thus, different decisions on which variables to revise first can lead to different DWO-revisions being discovered, which in turn can direct search tree to different parts of the search space with unpredictable results. Note that for structured problems only a few possible DWO-revisions are present in the revision list at each point in time, while for random ones there can be a large number of such revisions.

6 Conclusions

In this paper we showed how information about constraint weights can be exploited not only to perform variable selection, but also to order the revision list when arc consistency is applied during search. As a result, we proposed a number of simple and lightweight revision ordering heuristics for coarse grained arc consistency algorithms. The proposed heuristics order the revision list by trying to carry out possible DWO-revisions as soon as possible. Importantly, the heuristics can not only reduce the numbers of constraint checks and list operations but they can also have a significant effect on search. Among the heuristic we experimented with, the one with best performance was *dom/wdeg* in the variable-oriented implementation of arc consistency. Experimental results from various domains displayed the potential of the proposed heuristics.

Table 4. Cpu times (t), constraint checks (c) and nodes (n) from random problems using variable oriented propagation. Best cpu time is in bold.

Instance		*queue*	*dom*	*wdeg*	*d/wdeg*
frb30-15-1	t	26	19	26,7	**12,8**
	c	11,9M	8M	11,8M	5,4M
	n	6142	5648	6058	3659
frb30-15-2	t	69,4	**27,1**	108,3	86,6
	c	32,9M	15,7M	64,8M	49,6M
	n	18099	11617	36818	35822
frb35-17-1	t	114,6	176,5	**107,5**	228,6
	c	67,6M	103,6M	64,6M	130,2M
	n	27213	59585	28062	74098
rand-2-30-15	t	1130,1	**67,8**	89,3	98,5
	c	82,4M	38,2M	52,2M	56,2M
	n	42056	29056	29563	42115
geo50-20-d4-75-2	t	213,5	366,1	**31,7**	36
	c	138M	223,3M	20,3M	20,7M
	n	30747	88111	5468	8029

As future work, it would be interesting to study the interaction of revision ordering heuristics with other modern variable ordering heuristics apart from dom/wdeg. For example, the impact-based heuristics of [15] and the explanation-based heuristics of [9].

REFERENCES

[1] C. Bessière, E.C. Freuder, and J.C. Régin, 'Using Inference to Reduce Arc Consistency Computation', in *Proceedings of IJCAI'95*, pp. 592–599, (1995).

[2] C. Bessière and J.C. Régin, 'MAC and combined heuristics: two reasons to forsake FC (and CBJ?)', in *In Proceedings of CP-1996*, pp. 61–75, Cambridge MA, (1996).

[3] C. Bessière, J.C. Régin, R. Yap, and Y. Zhang, 'An Optimal Coarse-grained Arc Consistency Algorithm', *Artificial Intelligence*, **165**(2), 165–185, (2005).

[4] F. Boussemart, F. Hemery, and C. Lecoutre, 'Revision ordering heuristics for the Constraint Satisfaction Problem', in *10th International Conference on Principles and Practice of Constraint Programming (CP'2004), Workshop on Constraint Propagation and Implementation*, Toronto, Canada, (2004).

[5] F. Boussemart, F. Hemery, C. Lecoutre, and L. Sais, 'Boosting systematic search by weighting constraints', in *In Proceedings of 16th European Conference on Artificial Intelligence (ECAI '04)*, Valencia, Spain, (2004).

[6] A. Chmeiss and P. Jégou, 'Efficient path-consistency propagation', *Journal on Artificial Intelligence Tools*, **7(2)**, 121–142, (1998).

[7] M. van Dongen, 'AC-3_d an efficient arc-consistency algorithm with a low space-complexity', in *In Proceedings of the 8th International Conference on Principles and Practice of Constraint Programming (CP-2002)*, volume 2470, pp. 755–760, (2002).

[8] I.P. Gent, E. MacIntyre, P. Prosser, P. Shaw, and T. Walsh, 'The constraindedness of arc cosistency', in *In Proceedings of CP-97*, pp. 327–340, (1997).

[9] Cambazard H. and Jussien N., 'Identifying and Exploiting Problem Structures Using Explanation-based Constraint Programming', *Constraints*, **11**, 295–313, (2006).

[10] R.M. Haralick and Elliot, 'Increasing tree search efficiency for constraint satisfaction problems', *Artificial Intelligence*, **14**, 263–313, (1980).

[11] C. Lecoutre, F. Boussemart, and F. Hemery, 'Exploiting multidirectionality in coarse-grained arc cosistency algorithms', in *In Proceedings of the 9th International Conference on Principles and Practice of Constraint Programming (CP-2003)*, pp. 480–494, (2003).
[12] A. Mackworth, 'Consistency in networks of relations', *Artificial Intelligence*, **8**, 99–118, (1977).
[13] J.J. McGregor, 'Relational consistency algorithms anf their applications in finding subgraph and graph isomorphism', *Information Science*, **19**, 229–250, (1979).
[14] R. Mohr and T. Henderson, 'Arc and Path Consistency Revisited', *Artificial Intelligence*, **28**, 225–233, (1986).
[15] P. Refalo, 'Impact-based search strategies for constraint programming', in *In Proceedings of CP 2004*, pp. 556–571, (2004).
[16] D. Sabin and E.C. Freuder, 'Contradicting conventional wisdom in constraint satisfaction', in *In Proceedings of CP '94*, pp. 10–20, (1994).
[17] R. Wallace and E. Freuder, 'Ordering heuristics for arc consistency algorithms', in *AI/GI/VI*, pp. 163–169, Vancouver, British Columbia, Canada, (1992).

Penalties may have collateral effects. A MAX-SAT analysis.

Roberto Battiti and **Paolo Campigotto** [1]

Abstract.

Many incomplete approaches for MAX-SAT have been proposed in the last years. The objective of this investigation is not so much horse-racing (beating the competition on selected benchmarks) but understanding the qualitative differences between the various methods. In particular, we focus on *reactive search* schemes where task-dependent and local properties in the configuration space are used for the dynamic on-line tuning of local search parameters and we consider the choice between prohibition-based and penalty-based approaches. To abstract from implementation details we focus on the search trajectory characteristics and study the trade-off between diversification and bias after starting from a local minimizer. We then study the warping effects on the fitness surface of weight-update schemes and the resulting dynamics, through an exhaustive analysis of small MAX-SAT instances, and of the average evolution of individual trajectories. The results are compatible with the conclusion that penalty-based schemes achieve diversification from a starting local optimum through a complex method with global and potentially dangerous collateral effects, while prohibition-based schemes reach comparable or better results in a more direct and controllable manner.

In the final part of this paper, we consider long runs of the complete algorithms on selected MAX-SAT instances, which confirm the competitiveness of prohibition-based reactive approaches.

1 Introduction

Most of the incomplete methods based on stochastic local search (SLS) for MAX-SAT are characterized by a set of parameters whose tuning is crucial for CPU time requirements and solution quality. However, the appropriate tuning depends on both the problem and the current instance being solved, implying costly human intervention. Furthermore, the optimal parameter setting can vary widely in different regions of the configuration space around a given tentative current solution, leading to dynamic adaptive schemes. Reactive search strategies for the on-line dynamic tuning of these free parameters to the current task being solved and to the local characteristics can be used to obtain more robust and efficient techniques [1].

The scope of this paper does not allow a detailed review, see for example [2, 4] for a recent survey of propositional satisfiability and the related constraint programming problem, and [5] for a survey of stochastic local search approaches for MAX-SAT. Let us concentrate on reactive schemes and let us classify them according to the target acted upon during the on-line adaptation. In detail, the reaction can be on the generation of a set of *constraints on the variables* through the *prohibition* of recently-applied moves, or on the *modification of the cost function* guiding the local search. For brevity, the two paradigms will be denoted as *prohibition-based* and *penalty-based*, respectively. The first method, which is an ingredient of what is known as "tabu search," aims at pushing the configuration out from the attraction basin around a local minimizer by temporarily prohibiting some moves which would lead the trajectory back to the starting point. The second method, also termed "dynamic local search," modifies the objective function guiding the search so that a local minimum is raised to encourage the exploration of different areas, see Fig. 1.

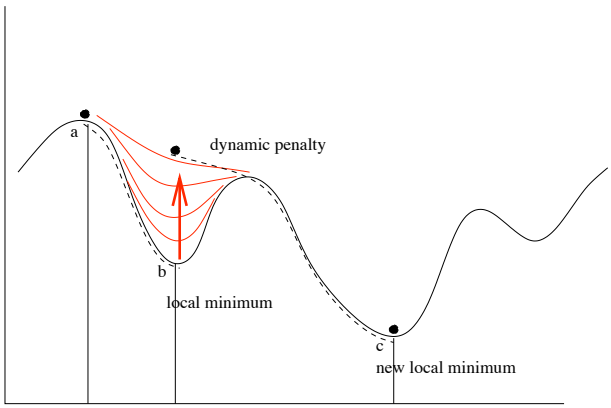


Figure 1. Transformation of the objective function to gently push the solution out of a given local minimum. Note: the intuition can be misleading for dynamically weighted clauses in MAX-SAT.

A second macroscopic difference is given by the selection of the variables considered during each local search step. In the basic schemes (like GSAT), all variables are potential candidates for the next flip, in more recent proposals (like WalkSAT), only the *variables appearing in unsatisfied clauses* are considered for a possible flip (let's call them "unsatisfied variables").

Given the space constraints of this paper we report selected results within an ongoing investigation to compare and identify qualitative differences between search dynamics of prohibition and penalty-based schemes.

[1] Dipartimento di Ingegneria e Scienza dell'Informazione (DISI), University of Trento, Italy, email: {battiti, campigotto}@disi.unitn.it

2 SLS approaches with on-line learning and dynamical systems

Let n and m denote the number of variables and clauses of a given MAX-SAT instance in conjunctive normal form. The simplest cost function guiding SLS algorithms for MAX-SAT is the number of unsatisfied clauses. This function will be denoted as f. Escaping local minima of f in a strategic and intelligent manner can be considered as the underlying motivation of most recent approaches based on stochastic local search. In many cases, local minima are actually large *plateaus* where the basic local search cannot determine the "right" direction to continue and performs a slow random-walk in search of an escape point. More advanced schemes design discrete dynamical systems so that the generated trajectory achieves a more efficient and effective exploration of the fitness surface. In this work we do not consider implementation details (supporting data structures) and CPU times but focus only on the trajectory properties. The fact that appropriate data structures make the advanced schemes fully competitive with the simpler ones has been matter of investigation but it is not included in this work due to space constraints.

A remedy to escape from local minima consists of transforming f into a modified g cost function, therefore *warping* the fitness surface, and generating a new direction of movement. This cost function modification may look at the *internal structure* of the current solution, not simply at the number of satisfied clauses. In [1] one exploits *non-oblivious* cost functions, which measure the *degree* of satisfaction of each clause by counting the number of matched literals. Aiming at a redundant satisfaction eliminates the difficulty in selecting among seemingly similar situations and may eventually permit to flip a variable to satisfy a new clause, without losing any already satisfied clause.

Another approach to escape from local minima or plateaus of f is given by *dynamic local search*, that relies on a dynamically weighted version of the oblivious function. The works in [8, 10] use dynamic weights to encourage the satisfaction of "more difficult" clauses. Clause weighting is motivated in [8] as a "breakout method for escaping from local minima", in [10] as a way to "fill-in" local minima. See for example [18, 15] and the contained references for some recent work in this area.

A starting point of this work is [17] where the authors investigate the dynamic warping of the search space caused by dynamic weight penalties, fail to find evidence that warped landscapes represent accumulated knowledge about the search space [3] and clarify that the "hole-filling" analogy can be deceiving by presenting a toy example showing the global and potentially detrimental side-effects, also hinted in [8]. Their empirical investigation shows that warping algorithms mainly serve as a diversification mechanism, which allows the search process to effectively overcome stagnation due to local minima and plateaus.

The purpose of this work is to continue the investigation through additional means:

Exhaustive analysis of warped landscapes Small (but non-trivial) instances of MAX-SAT are subjected to an exhaustive analysis of the modification (warping) effects by examining the local minima canceled, produced, and maintained after updating weights.

Diversification-Bias analysis It is suggested in [1] that Pareto-optimal points on D-B plots of basic versions of SLS schemes have an empirical predictive power for the overall success of the methods. This hypothesis is investigated for skeletal versions of prohibition- and penalty-based schemes.

Sample trajectories analysis While D-B plots summarize snapshots of the search after short periods starting from a local minimum event, and the exhaustive analysis describes the overall modification of the fitness surface, the analysis of sample trajectories produces additional information about the dynamical evolution of the search.

3 Exhaustive analysis of warped landscapes

First we perform an experimental analysis on a single unsatisfiable MAX-SAT instance formed by 20 variables and 110 clauses, close to the "satisfiability threshold region" [13].

We consider a prototypical and simplified version of the weight-update approach: as soon as the first local minimum (FLM) point for the "standard" f function is encountered, the weights of the currently unsatisfied clauses are increased by a fixed quantity Δw. An exhaustive analysis of the search space is then performed, showing the difference between the original and the warped fitness landscape generated by the weight update. The number of local minima of the landscape at different Hamming distances from the FLM point are counted. This classification allows to understand if the effects of the weight update are local or global, i.e., if the changes of the fitness landscape concentrate in the neighborhood region surrounding the FLM point or affect the whole search space. Due to the weight update operation performed when visiting the FLM point, new local minima can be generated in the search space (Fig. 2), while, at the same time, "old" local minima (i.e., local minima in the original landscape) may be canceled (Fig. 3). By definition, a local minimum disappears as soon as at least one improving move is available in its neighborhood.

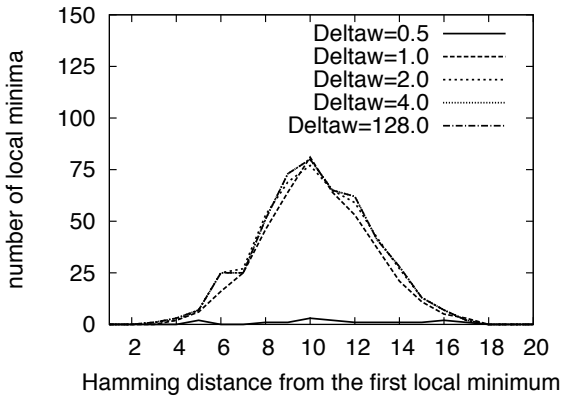


Figure 2. Distribution of the newly generated local minima over different warped landscapes. The curves describe the landscape generated by a weight increase operation with different Δw values $(0.5, 1, 2, 4, 128)$. The curve for $\Delta w = 0.5$ is at the bottom of the Fig.

The absolute numbers of local minima generated or canceled is bigger in the region around Hamming distance 10 from the FLM point. One suspects that the numbers are related to the original number of local minima present at specific distances, which is in turn related to the total number of binary strings at specific distances, a distribution peaked at distance $n/2$ with simple counting arguments.

Furthermore, the observed effects over the search landscape are not directly related to Δw: as soon as the Δw value is bigger than 1.0, very similar curves are obtained. The warped landscapes generated by the considered weight update values ranging from 4.0 to

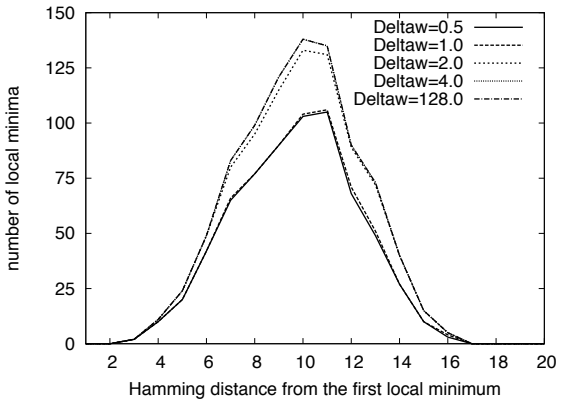


Figure 3. Distribution of the deleted local minima over different warped landscapes. The curves describe the landscape generated by a weight increase operation with different Δw values $(0.5, 1, 2, 4, 128)$. The curve for $\Delta w = 4.0$ overlaps with the curve for $\Delta w = 128.0$, while the curve for $\Delta w = 0.5$ and $\Delta w = 0.5$ are very close.

128.0 delete and generate exactly the same number of local minima at specific Hamming distances from the FLM point: the weighted clauses become so important that the effect of the other clauses is negligible.

Finally, the number of the canceled local minima is bigger than the number of the newly generated local minima.

Fig. 4 considers ratios instead of absolute numbers. Ratios underline that the deletion process of "old" local minima acts in a rather uniform way over the whole search landscape: it does not depend on the Hamming distance from the FLM point. Therefore, the changes affecting the search landscape in no way can be considered a localized effect.

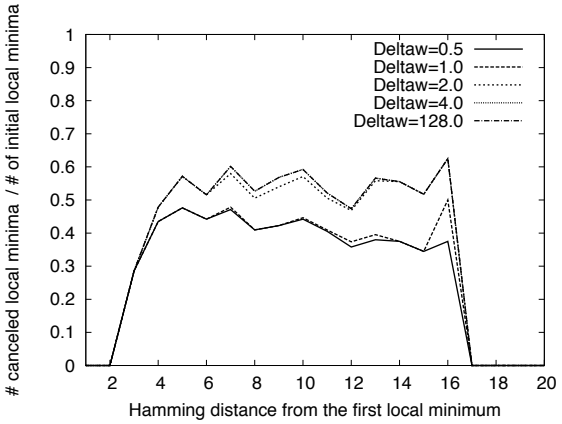


Figure 4. Fraction of canceled local minima over the search landscape. The curves are for the different Δw values $(0.5, 1, 2, 4, 128)$

The results clearly confirm the hints and the anecdotic evidence of previous researchers: weight updates caused by a specific local minimum encountered along the trajectory have a huge global effect: to escape from a single local attractor one is modifying in a radical manner also the value of configurations which are very far from the local minimum. In addition, a very large fraction of the initial local minima are canceled, about 50% in the given example, after an update caused by a single local minimum.

To measure the quality of the warped fitness landscape w.r.t. the original landscape, the quality of the deleted/generated local minima is also taken into account. The quality of the local minima is measured in terms of the cost function f, which counts the number of unsatisfied clauses. The smaller is the f value, the better is the quality of the local minimum, as it can be considered a better "approximation" of the global minima.

In particular, do weight-update schemes generate a "better" landscape, i.e., a landscape allowing better performance for local search strategies? This would be the case if poor quality local minima are ironed out but good quality ones are preserved. Furthermore, is the deletion/generation of the local minima related to their quality?

The weight-update mechanism aims at raising up (i.e., deleting) the local minimum which the algorithm is trapped in, allowing to escape from it (see Fig. 1). Let's call this local minimum the *current* local minimum. (Note that in our experiments the *current* local minimum is the FLM point).

There is a popular belief that the weight-update mechanism tends to delete low quality local minima in addition to the current one, obtaining a "cleaner" fitness surface. As a results, the warped surface should speed up the search of the global minima.

However, for the instance considered in the exhaustive analysis, we show this is not the case. Fig. 5 and 6 show the mean quality of the generated and deleted local minima w.r.t. the mean quality of the local minima in the original fitness surface. The different plots are for different weight update values shown in the labels. The plots for the weight update values 8.0, 16.0, 32.0 and 128.0 are equal to the plot for $\Delta w = 4.0$.

The quality of the newly generated local minima is worse than the quality of the deleted ones. Furthermore, for all the weight update values considered, the mean quality of the local minima in the warped surface never improves w.r.t. the mean quality of the local minima in the original surface (Table 1). In most of the cases, it is worse.

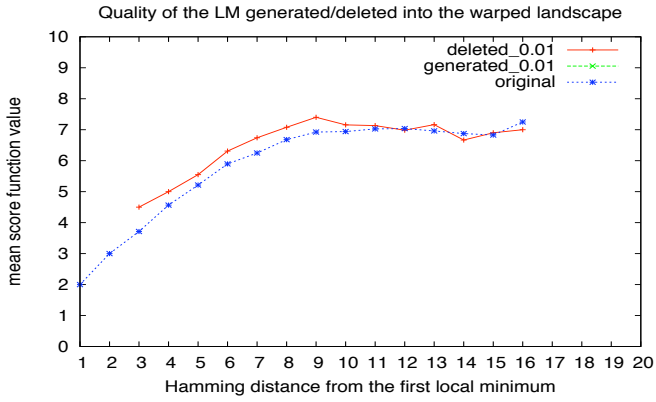
Δw	LM number	mean LM quality
0.0	1629	6.7114
0.01	958	6.5417
0.1	957	6.5423
0.5	974	6.5533
1.0	1383	6.8886
2.0	1244	7.0506
4.0	1227	7.1035

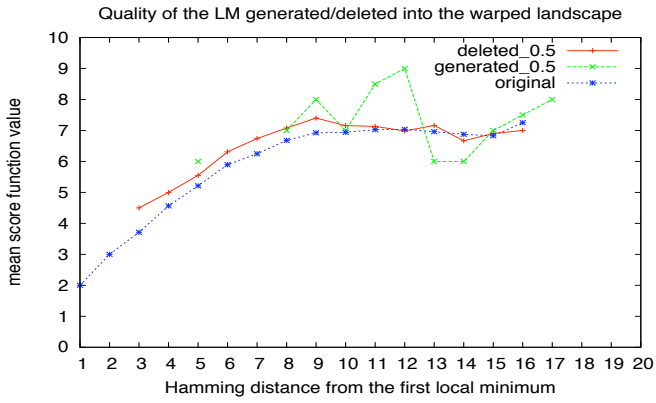
Table 1. Snapshot of the fitness surface after the weight-update operation. The first column indicates the Δw value, while the second and third ones show the number of local minima and their mean quality, measured in terms of the cost function f. The first line (weight update equal to 0.0) indicates the original "fitness surface".

Again, the strong global effect is observed: the quality of deleted/generated local minima does not depend on the Hamming distance from the FLM point.

In this experiment the *global* minima of the studied MAX-SAT instance (corresponding to 3 points with score function equal to 2) are not affected by the collateral effect of the weight update, i.e., they are still *global* minima in the warped surface.

The analysis performed shows that the warped surface obtained





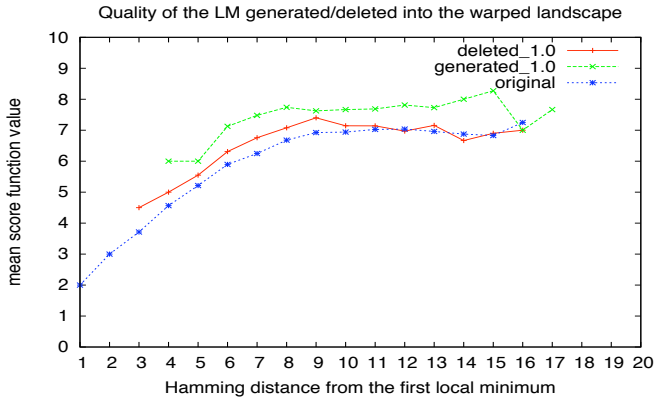
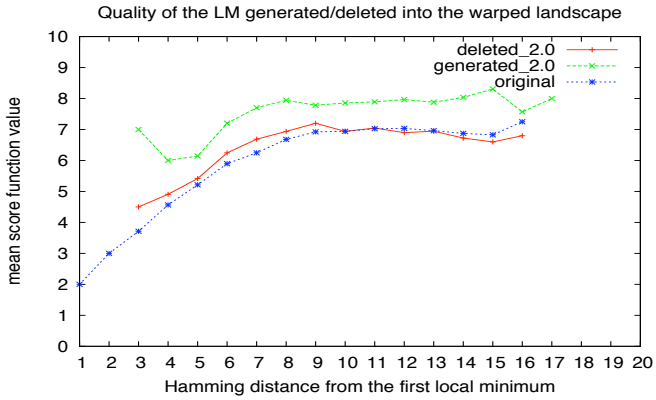
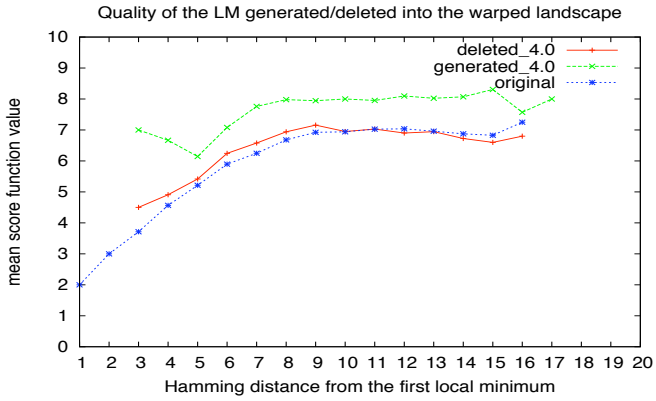


Figure 5. The mean quality of the generated/deleted local minima for the weight update values $0.01, 0.5, 1.0$. The curve labeled as "original" shows the mean quality of the local minima in the initial fitness surface.



(a)



(b)

Figure 6. The mean quality of the generated/deleted local minima for the weight update values $2.0, 4.0$. The curve labeled as "original" shows the mean quality of the local minima in the initial fitness surface.

in all the experiments does not correspond to one of a better quality for the local search techniques. The long runs of complete penalty-based algorithms reported in the last part of the paper fully validate this observation.

To verify that our observations are not biased by the MAX-SAT instances benchmark selected, we now consider a small hand crafted MAX-SAT instance obtained from the SAT 2005 competition, with 24 variables and 61 clauses, and repeat the exhaustive analysis of the search space. Fig. 7, 8 and 9 show the distribution of the deleted/generated local minima and the ratio among the deleted and the initial local minima, respectively.

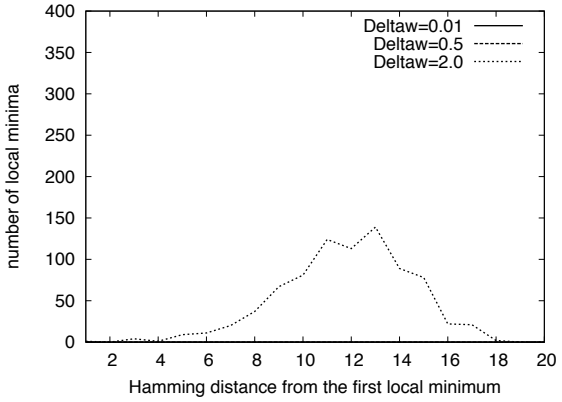


Figure 7. Distribution of the newly generated local minima over different warped landscapes. The curves describe the landscape generated by a weight increase operation with different Δw values $(0.01, 0.5, 2.0)$. For $\Delta w = 0.01$ and $\Delta w = 0.5$ no new local minima are generated.

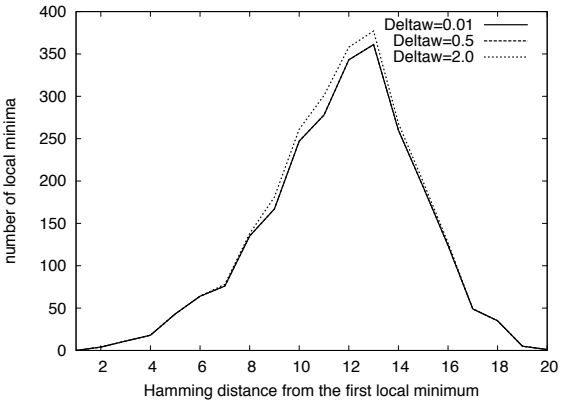


Figure 8. Distribution of the canceled local minima over different warped landscapes. The curves describe the landscape generated by a weight increase operation with different Δw values $(0.01, 0.5, 2.0)$

Fig. 10 compares the mean quality of the generated/deleted local minima w.r.t. the quality of the initial local minima for weight update values equal to 0.01, 0.5, and 2.0. The warped surface obtained for the weight update values $4.0, 8.0, 16.0, 32.0, 64.0$ and 128.0 is equal to the case $\Delta w = 2.0$.

Table 2 summarizes the changes among the original and the warped fitness surface for the various experiments.

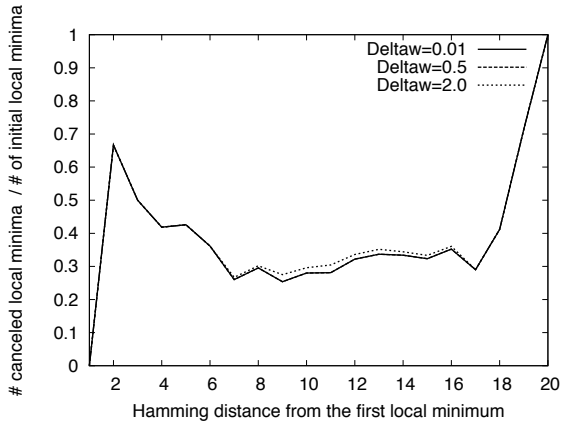


Figure 9. Fraction of canceled local minima over the search landscape. The curves are for the different Δw values $(0.01, 0.5, 2.0)$

Δw	LM number	mean LM quality
0.0	7756	3.7748
0.01	5341	3.5967
0.1	5341	3.5967
0.5	5341	3.5967
1.0	6139	3.7730
2.0	6056	3.7836

Table 2. Snapshot of the fitness surface after the weight-update operation. The first column show the weight update considered, while the second and third ones show the number of local minima and their mean quality, measured in terms of the cost function f. The first line (weight update equal to 0.0) indicates the original "fitness surface".

The results on the crafted instance confirm the observations for the random instance:

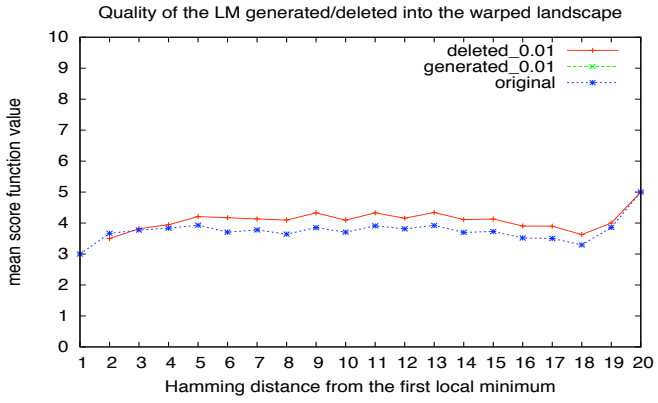
- the distribution and the quality of the deleted/generated local minima do not depend strongly on the Hamming distance from the FLM point;
- the number of the deleted local minima is bigger than the number of newly generated local minima;
- the mean quality of the local minima in the new fitness surface is never strongly improved (in some cases, it is even worse).

Again, the penalty-based approach exhibits a global and potentially undesiderable side effect.

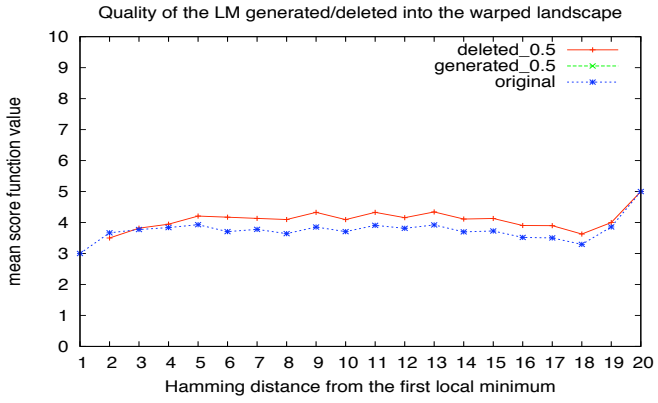
Furthermore, in this case the minima are affected by the weight- update operation. In the original fitness surface there are 134 minima; for the weight update values bigger than 2.0 considered, 12 of them are deleted: over the warped surface they are not even a local minimum. For the weight update values smaller than 1.0, one initial global minimum becomes a local minimum point in the warped surface, while 11 initial global minima do not remain local minima.

4 Diversification-bias analysis

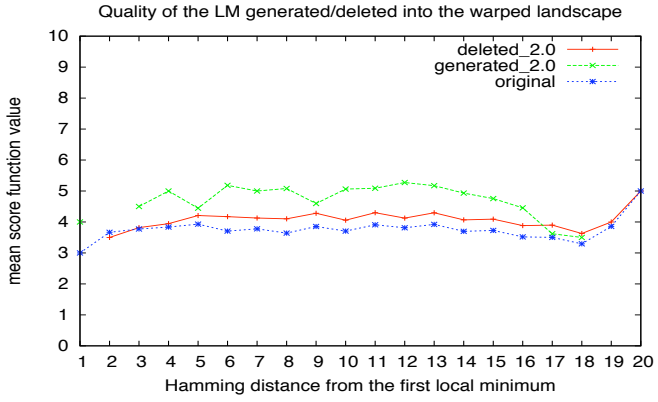
We follow the diversification-bias empirical analysis ("D-B plots") proposed in [1] where the authors conjecture that the dominant



(a)



(b)



(c)

Figure 10. The mean quality of the generated/deleted local minima for the weight update values $0.01, 0.5, 2.0$. The curve labeled as "original" shows the mean quality of the local minima in the initial fitness surface.

Pareto-optimal points on D-B plots of basic versions of SLS schemes have an empirical predictive power for the overall success of the methods. The metric used to measure the quality of the visited points (or, simply, the *bias* of the algorithm) is their average cost function value, while the diversification is measured via the average Hamming distance reached in short runs starting from a local optimum. A scheme (characterized by the choice of algorithm and parameter, Δw or prohibition T) is Pareto-optimal if there is no other scheme reaching both a higher diversification *and* a better bias. The prohibition scheme (GSAT/tabu) acts according to the very simple rule: after a single bit (truth value) is changed, it cannot be changed again for the next T iterations. Among the admissible bits (the ones which can be changed), one leading to the best Δf value is chosen randomly among the possible ties.

By using the D-B plots, we want to understand how the warped landscapes generated by the weight-update schemes affect the performance of the dynamic local search (DLS) algorithms, considering both bias and diversification. In particular, we compare the results for the weight-update method with the results of the prohibition-based approach.

All runs of the algorithms considered proceed as follows: as soon as the first local optimum for the "standard" f function is encountered, it is stored and the algorithm is then run for additional $4 * n$ iterations. The final D-B values averaged over 500 tests are reported. For each test we identify the first local minimum via the GSAT algorithm, and then, depending on the different test, we run one among GSAT/tabu and the weighted version of GSAT, starting from the the discovered local minimum.

The tests presented in this work are dedicated to selected MAX-3-SAT instances defined in [7]. In detail, if $n : m$ identify variables and clauses, 50 instances for the 20:110 cases have been randomly generated. The different algorithms are run for the different instances, for a total of 500 tests. The average results are presented.

First, we evaluate the GSAT/tabu method based on fixed prohibitions. Then we study the performance of a simplified version of the weighted GSAT algorithm. Initially all clause weights are equal to one and, once the first local optimum is encountered, the weights of the currently unsatisfied clauses are increased by a fixed quantity Δw. Let's note that the increment of the weights is performed only once: for the subsequent local minima, weights remain fixed.

Fig. 11 shows the results obtained by running the considered algorithm for $4 * n$ steps after the first local minimum discovered by the GSAT algorithm for the same SAT instances. The labels for the curve named *gsatWeighted* represent the different Δw values considered. The value 0 for the "gsatWeighted" curve represents the case of the original GSAT algorithm [12]. The curve named *gsatTabu* is labeled with the values for the fractional prohibition T_f, given by the prohibition parameter divided by n.

The bias is plotted as *difference* w.r.t. the starting f value at the local minimum (good values are therefore at the bottom), the Hamming distance is divided by the number of variables n. One notes that small prohibitions values lead to bias levels comparable with the penalty-based scheme, but they allow for a bigger diversification. This result is confirmed by the repetition of the experiment over the 500:5000 instances, which shows a bigger set of Pareto-optimal points for the prohibition-based scheme (Fig. 12).

In Sec. 6 we consider the original complete schemes for penalty-based and prohibition-based approaches, including the reactive and dynamic versions and analyze the average f values obtained for long runs. The D-B plots Pareto-optimal points are indeed accurate predictors of performance over the long runs.

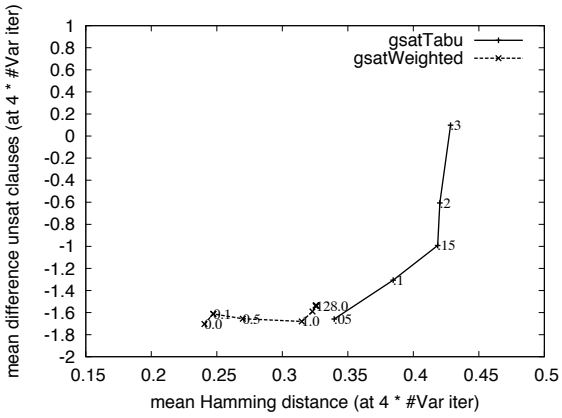


Figure 11. Diversification-bias plots of prohibition-based and penalty-based strategies (20:110 MAX-SAT instance). The curve named *gsatTabu* is labeled with the values for the fractional prohibition T_f, the curve named *gsatWeighted* is labeled with the Δw values for the weights update (points for $\Delta w = 8, 16, 32, 64, 128$ are at a very similar position).

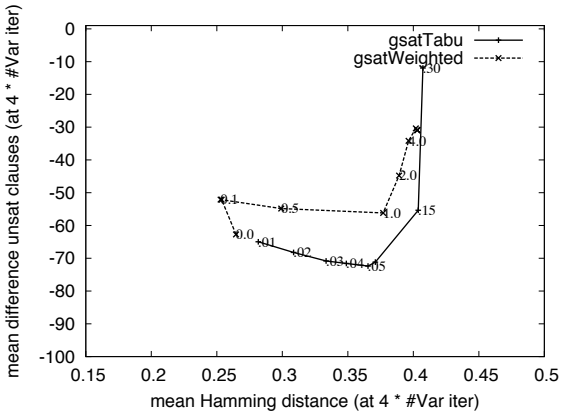


Figure 12. Diversification-bias plots of prohibition-based and penalty-based strategies (500:5000 MAX-SAT instance).

5 Sample trajectories analysis

We now consider the impact of the penalty-based and of the prohibition-based mechanisms on the dynamical evolution of the search trajectories. In particular, do the changes on the landscape caused by the weight-update strategy significantly affect the behavior of a dynamic local search algorithm? If yes, is the biased behavior observed while visiting the region surrounding the first FLM point or over the whole search landscape? Furthermore, we ask whether the prohibition-based strategy, that does not modify the fitness surface, performs a similar function but with a more direct manner. Let's consider again the simplified prohibition-based mechanism and the weight-update strategy described so far. We execute a single short run of both methods starting from a FLM point on the 20:110 instance considered (the first point encountered by local search starting from an initial random configuration), and measure the diversification level reached in both cases (Fig. 13 and 14). We repeat the experiment using different values of the fractional prohibition T_f and of the Δw quantity. The trajectories of the prohibition-based scheme

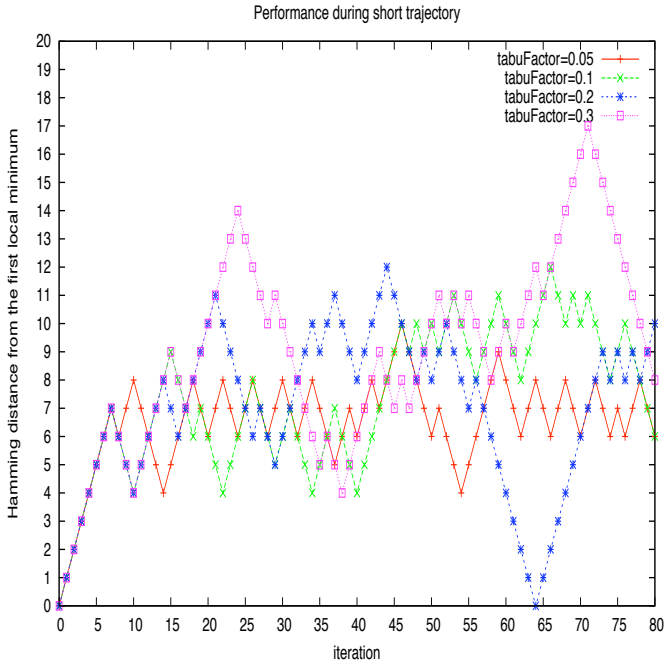


Figure 13. Evolution of the Hamming distance of single short trajectories of the prohibition-based strategy .

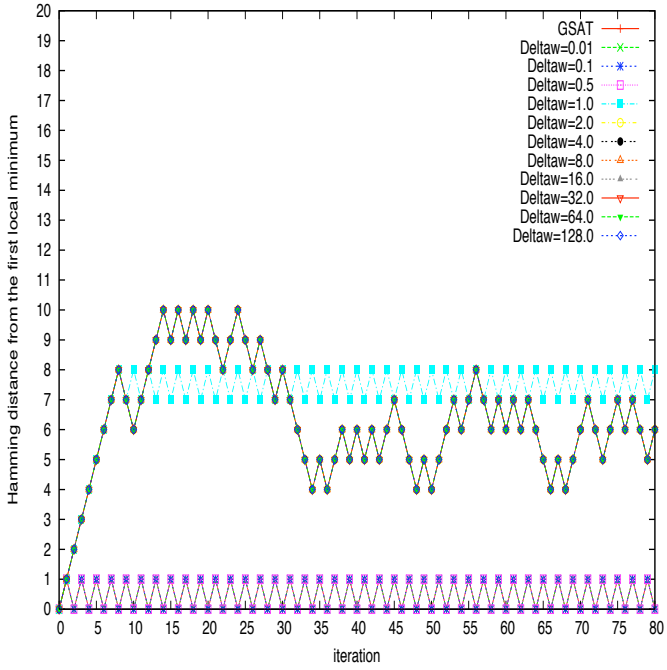


Figure 14. Evolution of the Hamming distance of single short trajectories of the penalty-based strategy. The curves for $GSAT$ and Δw values smaller than 1.0 overlap (bottom of the Fig.), and the curves for Δw values bigger than 1.0 are the top ones overlapping.

show an initial Hamming distance growing linearly up to distance $T+1$, a deterministic effect cause by the prohibition mechanism, followed by a diverse and randomized exploration of the search space. The average deviation of the distances tends to grow with the larger prohibition values. No evidence of entrapment is shown in the later steps.

The situation is qualitatively different for the penalty-based scheme. For small Δw values (values smaller than 1.0, in this case), the trajectory shows a cycle of length 2. For bigger values, the algorithm escapes from the attraction basin of the FLM point, but it is eventually stuck at another local minimum. For the 1.0 value, the algorithm cannot escape from this second local minimum. Furthermore, for all the values bigger than 1.0 the same trajectory is observed. This is not surprising, as in the exhaustive analysis we observed that the number of canceled/generated local minima is the same in all cases. To be fair, let's note that the observed cycles may be avoided by considering more complex weight-update based mechanism, that perform the weight update for each local minimum encountered during the trajectory.

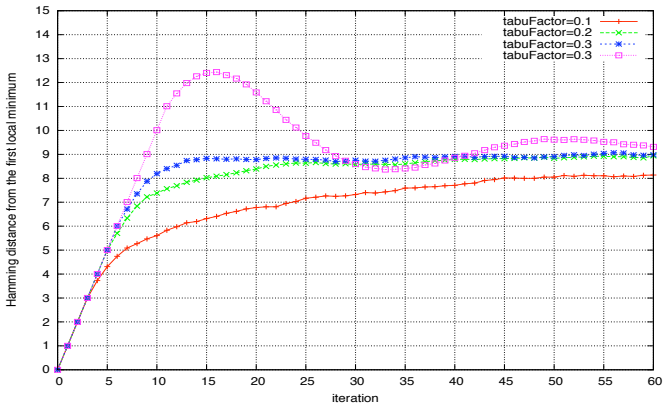


Figure 15. Performance of prohibition-based strategy during short trajectories. The values are averaged over 1000 trajectories starting at a different initial FLM point.

To validate our initial observation of sample trajectories, we now consider the performance of penalty-based and prohibition-based approaches averaged over 1000 runs performed starting from 1000 different initial local minima. Fig. 15 show that during the first iterations of the prohibition-based approach the diversification strictly increases, as expected, and that, eventually, it tends to converge to a common value. The memory about the initial local minimum is effectively lost and exploration proceeds without hindrance. Furthermore, larger values for the prohibition T lead to a bigger initial diversification. Fig. 16 clearly indicates that the performance in terms of diversification of the penalty-based approach is worse than that of the prohibition-based strategy: smaller Hamming distances are reached and the effect shows a fragile dependence on the Δw values, which can be compared to the rather similar behavior of different T values after runs of comparable length (60 iterations in our case). As suggested by intuition, a better diversification is reached with the bigger Δw values. In particular, the worst performance is reached by the GSAT algorithm, that operates over the non-weighted f function.

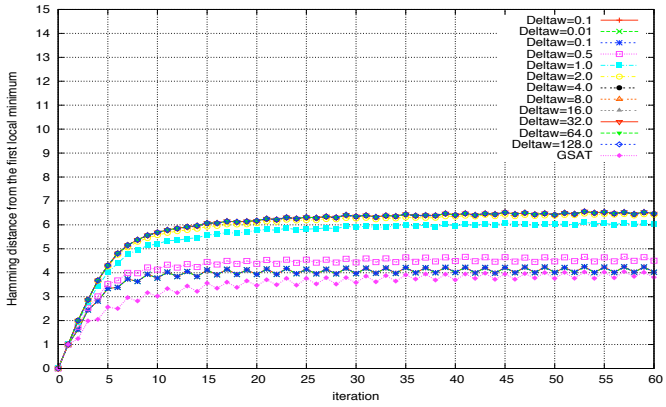


Figure 16. Performance of penalty-based strategy during short trajectories. The values are averaged over 1000 trajectories starting at a different initial FLM point.

6 Experiments on long runs

Even if this work does not target the horse-racing point of view, to validate the results of the exhaustive analysis and of the D-B plots experiments, we show the MAX-SAT results reached by the penalty-based and the prohibition-based approaches. In particular, we consider the 500:5000 and the 300:1500 MAX-SAT benchmarks and execute the following SLS approaches:

- GSAT [12], a basic local search greedy strategy guided by the score function f, that simply counts the number of unsatisfied clauses;
- GSAT/tabu [14], which enriches the GSAT algorithm via a prohibition-based search criterion;
- WalkSAT/SKC [11], the ancestor of the WalkSat family. It randomly alternates between greedy minimizing moves and random noisy moves. The moves of both kinds act on the variables appearing in unsatisfied clauses;
- WalkSAT/tabu [6], that adopts the same score function and the same variables selection mechanism of the WalkSAT/SKC algorithm, complemented by tabu search;
- H-RTS, a Hamming-based reactive tabu search algorithm, that dynamically adapts the prohibition parameter during the search;
- AdaptNovelty$^+$ [16], that exploits the concept of variable "age" and uses the same scoring function of GSAT. The variable age can be considered a sort of soft prohibition of recently-changed variables in the case of ties. The prefix *"Adapt"* underlines a reactive behavior, that dynamically adjusts its internal parameters;
- Scaling and Probabilistic Smoothing (SAPS) [18], an accelerated version of the Exponentiated Subgradient algorithm [9] based on dynamic penalties, and a reactive version thereof called RSAPS.

For brevity we report here only the average results (10 runs with different random seeds for each of the 50 instances) as a function of the number of iterations (flips). The user of SLS algorithms is typically interested in the number of iterations required by each algorithm to reach the desired results, or, at least, a good quality approximation. As predicted by the previous diversification-bias analysis and according to the exhaustive search experiments performed, the curves in Fig. 17 confirm a clear superiority of the prohibition-based techniques with respect to the penalty-based approaches. The

error bars are not shown on the plots to avoid cluttering. Among all the possible values for the tabu parameter of the WalkSAT/tabu algorithm, we plot the case where the fractional prohibition T_f is 0.01, as with this setting we obtain the best performance over the considered benchmark. The same for the GSAT/tabu algorithm, whose curve is drawn for the optimal T_f value 0.05 over our benchmark set.

The SAPS parameters have been set to the default values, without attempting any extensive optimization. Preliminary tests obtained changing the values did not lead to significant improvements.

With this optimal setting, the GSAT/tabu algorithm reaches eventually a performance equivalent to that of H-RTS, even if its performance is inferior in the initial phase. This result clearly indicates that parameters setting is crucial for the algorithms performance: not only H-RTS reaches results comparable to the ones with a fixed and optimal T_f, but it actually improves on these because of the dynamic on-line adaptation. This observation is emphasized by the curves for SAPS and RSAPS. They confirm the effectiveness of the reactive approach, that obtains better results while, at same time, allowing to avoid the manual tuning. The SAPS parameters have been set to the default values, without attempting any extensive optimization. Preliminary tests obtained by changing the values did not lead to significant improvements.

Finally, the curve for H-RTS shows the effectiveness of the NOB search to rapidly discover good local optima.

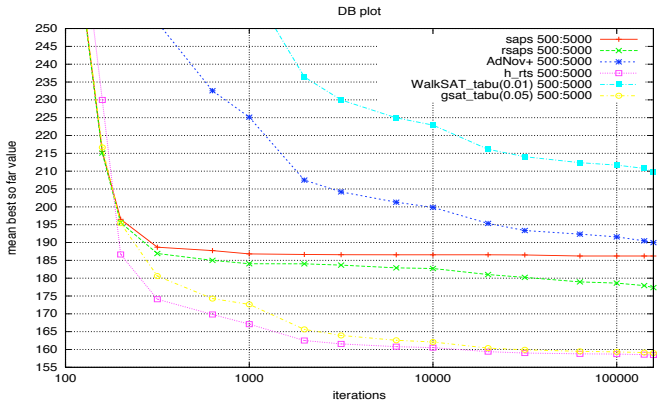


Figure 17. The mean best so far bias value reached by the SAPS, RSAPS, AdaptNovelty$^+$ and H-RTS algorithms.

Fig. 18 shows the behavior of the same algorithms in a scenario closer to the satisfiability threshold (the clauses/variables ratio of the 300:1500 tasks is 5) and to the small 20:110 MAX-SAT instance considered for the exhaustive analysis. The results of the previous long runs experiment are confirmed, apart from the competitive performance of AdaptNovelty$^+$ which eventually duplicates H-RTS' performance although with a much slower start.

Let us note that many of the considered techniques have been proposed for SAT and one may argue that a direct comparison with MAX-SAT algorithms such as H-RTS is not fair. On the other hand, the underlying logic of the methods is always based on maximizing the number of satisfied clauses, which is an argument in favor a direct comparison, in particular for adaptive techniques. In any case, this issue will be explored further in the future.

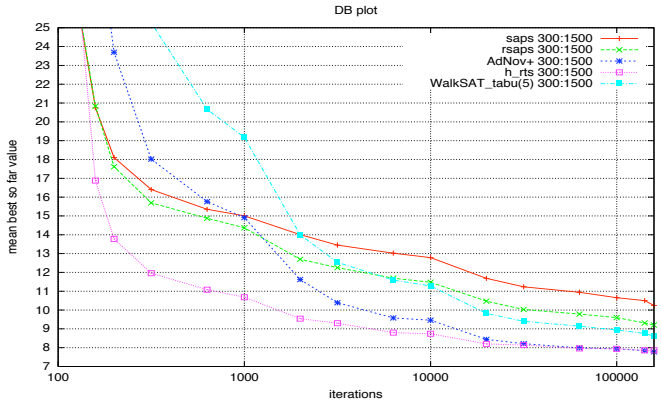


Figure 18. The mean best so far bias value reached by the SAPS, RSAPS, AdaptNovelty$^+$ and H-RTS algorithms on 300:1500 instances.

7 Conclusion

We presented some selected results of an ongoing comprehensive evaluation of alternatives design strategies for MAX-SAT algorithms base on stochastic local search. In particular, we focused on studying the qualitative differences of the dynamics caused by prohibition- and penalty-based schemes, by exhaustively analyzing the warped landscape of small problems, by measuring the diversification and the bias after starting from local minima, and the average behavior of sample trajectories.

The results confirm the hypothesis that penalty-based modifications of the search landscape have global side-effects with a potentially disturbing influence on the search trajectory. The real advantage of these schemes appears to be the fact of forcing the trajectory to abandon an area around a local optimizer to avoid confinement.

On the other hand, a very similar "escaping" effect can be obtained by direct prohibition-based schemes, which do not require the addition of explicit forgetting schemes as a cure to the potentially harming side-effects and are more amenable to explanation. We are aware of the preliminary and in part controversial nature of this investigation, which motivates additional work to further substantiate this hypothesis, when both dynamical system properties and the final competitiveness of the implemented schemes are considered.

REFERENCES

[1] R. Battiti and M. Protasi, 'Reactive search, a history-sensitive heuristic for MAX-SAT', *ACM Journal of Experimental Algorithmics*, **2**(ARTICLE 2), (1997). http://www.jea.acm.org/.

[2] L. Bordeaux, Y. Hamadi, and L. Zhang, 'Propositional Satisfiability and Constraint Programming: A comparative survey', *ACM Computing Surveys (CSUR)*, **38**(4), (2006).

[3] J. Frank, 'Learning short-term weights for GSAT', in *Proc. of the Fifteen INTERNATIONAL JOINT CONFERENCE ON ARTIFICIAL INTELLIGENCE*, volume 15, pp. 384–391. LAWRENCE ERLBAUM ASSOCIATES LTD, USA, (1997).

[4] C.P. Gomes, H. Kautz, A. Sabharwal, and B. Selman, *Handbook of Knowledge Representation.*, chapter Satisfiability solvers, Elsevier, 2008, in press.

[5] H. H. Hoos and T. Stuetzle, *Stochastic Local Search: Foundations and Applications*, Morgan Kaufmann, 2005.

[6] D. McAllester, B. Selman, and H. Kautz, 'Evidence for invariants in local search', in *Proceedings of the national conference on artificial intelligence*, number 14, pp. 321–326. John Wiley & sons LTD, USA, (1997).

[7] D. Mitchell, B. Selman, and H. Levesque, 'Hard and easy distributions of SAT problems', in *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI-92)*, pp. 459–465, San Jose, Ca, (July 1992).
[8] P. Morris, 'The breakout method for escaping from local minima', in *Proceedings of the national conference on artificial intelligence*, number 11, p. 40. John Wiley & sons LTD, USA, (1993).
[9] D. Schuurmans, F. Southey, and R.C. Holte, 'The exponentiated subgradient algorithm for heuristic boolean programming', in *Proceedings of the international joint conference on artificial intelligence*, volume 17, pp. 334–341. Lawrence Erlbaum associates LTD, USA, (2001).
[10] B. Selman and H.A. Kautz, 'An empirical study of greedy local search for satisfiability testing', in *Proceedings of the eleventh national Conference on Artificial Intelligence (AAAI-93)*, Washington, D. C., (1993).
[11] B. Selman, H.A. Kautz, and B. Cohen, 'Noise strategies for improving local search', in *Proceedings of the national conference on artificial intelligence*, volume 12. John Wiley & sons LTD, USA, (1994).
[12] B. Selman, H. Levesque, and D. Mitchell, 'A new method for solving hard satisfiability problems', in *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI-92)*, pp. 440–446, San Jose, Ca, (July 1992).
[13] Bart Selman, David G. Mitchell, and Hector J. Levesque, 'Generating hard satisfiability problems', *Artif. Intell.*, **81**(1-2), 17–29, (1996).
[14] Olaf Steinmann, Antje Strohmaier, and Thomas Stutzle, 'Tabu search vs. random walk', in *KI - Kunstliche Intelligenz*, pp. 337–348, (1997).
[15] John Thornton, Duc Nghia Pham, Stuart Bain, and Valnir Ferreira Jr., 'Additive versus multiplicative clause weighting for sat', in *AAAI*, pp. 191–196, (2004).
[16] Dave A. D. Tompkins and Holger H. Hoos. Novelty$^+$ and adaptive novelty$^+$. SAT 2004 Competition Booklet, 2004. (solver description).
[17] Dave A. D. Tompkins and Holger H. Hoos, 'Warped landscapes and random acts of sat solving', in *Proc. of the Eighth International Symposium on Artificial Intelligence and Mathematics (AIMA-04)*, (2004).
[18] F. Hutter D.A.D. Tompkins and H.H. Hoos, 'Scaling and probabilistic smoothing: Efficient dynamic local search for sat', in *Proc. Principles and Practice of Constraint Programming - CP 2002 : 8th International Conference, CP 2002, Ithaca, NY, USA, September 9-13*, volume 2470 of *LNCS*, pp. 233–248. Springer Verlag, (2002).

Debugging Constraint Models with Metamodels and Metaknowledge [1]

Eugene C. Freuder, Richard J. Wallace and Tomas E. Nordlander
Cork Constraint Computation Center and Department of Computer Science
University College Cork, Cork, Ireland
email: {e.freuder,r.wallace}@4c.ucc.ie,tomas.nordlander@sintef.no

Abstract. A major challenge in constraint programming is to create an adequate constraint model for a given problem. This is an iterative process, in which an original model must be amended or 'debugged'. An important source of guidance in this effort is the existence of solutions not allowed by the model in its current form but which the user either knows are solutions or wants to have included. This paper describes a system, "Ananke", which allows this aspect of the modelling process to be partly automated. In Ananke, the current model is considered an instantiation of a metamodel whose solutions are sets of constraints that together form a CSP. In this system, given a CSP model, the user suggests a partial or complete solution. If that solution is not part of the current model, the system uses its metamodel to search for changes in constraints that would produce a revised model containing the proposed solution. In this work, we consider different metrics for judging quality of revisions, including the number of additional tuples that will result from a change and number of additional solutions. We also consider heuristics that improve search efficiency, including some that limit the constraints that may be changed. This work shows how meta-reasoning can be used effectively to support the process of developing viable constraint satisfaction models.

1 INTRODUCTION.

A CSP model may be incomplete or incorrect. In this situation we would like the computer to provide "debugging" assistance in completing or correcting the model. This raises very general questions as to what it means for a model to be complete and correct, what kinds of assistance users would find helpful, and how such assistance might be efficiently and effectively provided. Here, we describe a specific form of assistance to address a specific class of model deficiencies:

- the user indicates that there should be a set of solutions with a specified property,
- the system indicates that there is not and suggests alterations to the model that will admit such solutions,

and we develop and evaluate procedures for providing this assistance.

We will utilize a very simple temporal reasoning problem to illustrate some of the basic concepts here. There are three meetings to be scheduled, each can be held at 11 or 1, meeting A must be held before meeting B, and meeting B must be held at the same time as meeting C. A specific interaction here might be:

- User: There should be at least one solution where meeting A can be held in the afternoon.
- System: There are no such solutions at present. Currently, meeting A is constrained to occur before meeting B. If we changed that to allow it to occur after meeting B, we would have a solution with meeting A in the afternoon. Shall we do that?

This form of interaction, which addresses overconstrained problems, might be viewed as complementary to the interaction in the Matchmaker system [4] where the system suggests solutions to underconstrained problems and the user indicates that these are not solutions and provides alterations to the model to prohibit them. The approach we take here could also be extended to underconstrained problems, or problems that are simultaneously over and under constrained. This form of interaction is also directly related to a considerable body of work on modeling, debugging, configuration, interactive CSP, dynamic CSP, partial CSP, hierarchical CSP, soft CSP, explanation, compilation, acquisition and learning within the CP community and more broadly to work in knowledge acquisition, knowledge engineering, truth maintenance, machine learning, and diagnosis.

In particular, the QuickXPlain algorithm [5] for determining preferred relaxations and the CONACQ algorithm [2] for acquiring constraint networks from training sets might be viewed as already addressing specific forms of this general interactive paradigm, and might be usefully extended or combined in this context. However, we wish to explore the possibility that "bespoke" solutions to specific forms of debugging may provide additional efficiency and flexibility. The underlying premise here is that it can be worthwhile to identify specific special cases of the general debugging challenge, and specific methods for dealing with them, then work to generalize, adapt, or combine these methods.

This work is part of a broader research programme to develop a mixed-initiative interface that can provide a variety of assistance modes to address in a user-friendly fashion the variety of knowledge acquisition, validation, and revision issues that can arise in defining, debugging, and maintaining models for constraint satisfaction and optimisation applications. This work is inspired in part by the classic early work on knowledge acquisition and debugging in a rule-based context, Teiresias. We have named our project Ananke, after the 'Goddess of Bonds', the deity that rules constraint and coercion.

The next section, 2, discusses metaknowledge and metamodels. Section 3 presents an overview of the present implementation of Ananke. Section 4 describes Ananke's debugging procedure in more detail. Section 5 gives results of some preliminary tests of the sys-

[1] This work received support from Science Foundation Ireland under Grant 05/IN/I886. We thank Benne Jakobus for help with some of the coding.

tem. Section 6 discusses related work. Section 7 gives conclusions and a brief note on future prospects.

2 METAKNOWLEDGE AND METAMODELS

We start with the basic model of a CSP, as a triple (X, D, C), where X is a set of n variables, D is a multi-set of domains in 1:1 relation with X, and C is a set of constraints applied to X. Also, an assignment is a set of elements each drawn from a separate domain in D; if the assignment is of size n and satisfies all the relations in C, then it is a solution to the CSP model.

2.1 Metaknowledge

We focus our study here by making the following assumptions:

- The property that characterizes the "missing" solutions can be expressed as a set of additional "missing solution constraints" (ms-constraints) over a subset of the problem variables, the ms-variables. We will say that the ms-constraints are a form of metaknowledge. In the example above, the ms-constraints would consist of the single constraint that meeting A occurs in the afternoon, and ms-variables the single variable corresponding to meeting A.
- For each constraint in the original problem there is a set of alternative constraints to choose from. In the example, we could specify a different form of relation between meetings A and B. Or we could suggest removing the constraint entirely.

More formally, a missing solution constraint, C_{ms} can be defined as a set-constraint whose tuples are the possible solution sets to the problem. The possible values of this constraint are, therefore, the powerset of the set of tuples allowed by a set of CSP variables that are (partial) solutions. This can be written:

$$\{\{(X_1/v_1, X_2/v_2 \ldots X_k/v_k)\}\},$$

where X_1, X_2 $\ldots$ X_k are the k variables in the assignment, and $k \leq n$, the number of variables in the CSP, and $v_1, v_2 \ldots$ are assignments to these variables that form a viable (partial) solution. The constraint itself consists of all such solution sets that contain the missing solutions.

This formulation allows for the fact that the user may not know complete missing solutions, or that an explicit list might be too large to enumerate. It also allows specifications in terms of subsets of domains, and this allows the user to characterize what is missing more abstractly than by simple enumeration.

Our goal here is to assist the user in exploring the space of possible models for the problem under consideration. In the above example, the user may be saying "I am an expert and I know what solutions to expect; if this program doesn't provide them it has the wrong model" or the user may be saying "If the model does not allow this type of solution, I am willing to change the model".

2.2 Metaproblems

We define a constraint metaproblem, or MCSP, as a CSP whose solutions are CSPs (viewed as sets of constraints). The values of the metaproblem variables, or metavariables, will be constraints. In the example, an MCSP could consist of 5 metavariables, 3 corresponding to the unary constraints specifying the domains of the 3 meeting variables and 2 corresponding to the temporal constraints, and one ms-constraint, which in this case is also unary. The domain of values for the latter could be the possible equality/inequality relations between two quantities, or it could be the Allen's algebra constraints. The former each have a single unary constraint value, specifying the set $\{11\ 1\}$. A ground solution to an MCSP is defined to be a solution to one of the CSPs that is itself a solution to the MCSP.

In general we can have MCSP constraints, or metaconstraints, on the metavariables; e.g. an equality metaconstraint between the metavariables corresponding to the two temporal constraints in the example could specify that these constraints have to be the same. This provides a mechanism for future elaboration of the interaction between user and system. Our current implementation, however, only considers unary metaconstraints.

Given a CSP we can now define a specific form of the general user/system interaction paradigm we started with:

- The user specifies that there should be a set of solutions to the CSP augmented by a set of ms-constraints.
- The system indicates that there is not and suggests alternative choices for some of the constraints that will admit such solutions.

Our problem now is to derive these suggestions for changing constraints. We can formulate this problem itself as a kind of CSP, in terms of a metaproblem, the missing solution CSP or MS-CSP. This is a bit baroque to define precisely, but the underlying idea is straightforward: We have to choose alternative constraints for some of the constraints that allow us then to solve the resulting problem, augmented by the ms-constraints.

Specifically, we construct the MS-CSP as follows:

1. Take the original CSP. Let us use our running example.
2. Replace the constraints with metavariables. In the example, the constraint between A and B would be replaced by the metavariable corresponding to this constraint, whose domain of values consisted of the possible relations that could hold between these variables.
3. Add the ms-constraints as new metavariables. However, instead of implementing these constraints according to the basic definition, we include only the tuples formed by the user's specifications. In the present example, the new metavalue would be a unary constraint on meeting A requiring it to be in the afternoon.

The argument for implementing ms-constraints in this way is straightforward. We know that the set of solutions of the original CSP does *not* satisfy the ms-constraint. However, any selection of values for the metavariables consistent with the tuple-set specified by the user will be included in the set of solution-sets that satisfies the ms-constraint. This is because the set of partial assignments is a subset of the intersection of all solution sets that contain this set as a subset, i.e.

$$\{(specif.\ assign.)\} \subseteq \bigcap \{\{solns\ incl.\ specif.\ assigns.\}\}$$

If, in addition, we choose values for the metaconstraint variables that serve to relax the original CSP constraints, we will not lose any of the original solutions. The present implementation is organised to make selections according to this requirement.

2.3 Solutions and Metrics

Finding ground solutions to the MS-CSP involves choosing both metavariable values, i.e. constraints, and variable values. We can make these choices with some form of backtrack search, as described

below. This raises questions of efficiency, but also of the quality of solutions, since there are many alternative ways that quality could be measured.

Metrics are required for searching through the set of possible CSP models efficiently. A natural consideration for judging the quality of suggestions is the amount of change they represent from the original model. Perhaps the most basic metric of this kind is the number of solutions that differ between the given and modified model. But, since computing all solutions is expensive, we will consider other cruder metrics of this type.

The simplest is to arrange the possible constraints into a partial order, in which any change that relaxes a constraint, i.e. allows more tuples to satisfy it, is judged as having a greater cost than a tighter constraint. Then, single steps in the resulting lattice can be considered to add a cost of 1 to the overall cost of the change.

A more sophisticated metric that is still often simple to compute is based on the number of tuples that would be added if the constraint were relaxed in a certain way. For example, if a binary $>$ constraint were relaxed to $\geq$, this would add one tuple, while if it were relaxed to $\neq$, then $d^2 - d$ tuples would be added. This form of metric can be cast in a general form: a lattice whose elements are associated with a number of tuples ranging from 0 to d^k, where k is the arity of the constraint.

2.4 Heuristics

In addition to guidance from the chosen metric, it is possible to use various heuristic methods to make search more efficient. A basic hypothesis here is that the metaknowledge can help direct the search. These methods, however, often entail tradeoffs between efficiency and quality of the resulting suggestion.

The most important heuristic is one that orders the changes in a given constraint according to the cost function employed. In many cases, changes can be preordered even if the exact cost is not known ahead of time. For example, a $\geq$ constraint will always allow at least as many tuples as a $>$ constraint. Employing this heuristic greatly enhances the bounding process in branch and bound search (in the monotonic context), because once the current bound has been exceeded for a given constraint change, then any more expensive change will also exceed the bound.

Another heuristic method is to restrict search to constraints linking the ms-variables and other problem variables, which we will call "link constraints." In this case, we can 'fix' the problem by just changing or removing one or more link constraints. In the example, we changed the link constraint between meeting A and meeting B. Obviously, if one of the possible changes is removing a constraint, this method will always produce a solution. However, it may not be as desireable as the best-cost solution obtained by searching through the complete space of possible changes. On the other hand, if one of the criteria for goodness of a solution is that the changes occur as close to the failed assignment as possible, then this criterion can be incorporated in the cost function.

Link constraints can be found by doing an all-solutions search with the original CSP model while keeping track of solutions that match the desired solution or solutions in as many assignments as possible. Here, there are several alternative approaches that can be used to specify link constraints:

- taking the constraints adjacent to the unassigned variables in an arbitrary best-match, e.g. the first best-match found during search,
- taking the intersection of unassigned variables in all best-match solutions,
- taking the union of the unassigned variables in all best-match solutions.

Another straightforward method is to include the metaknowledge regarding missing solutions in order to guide the search for suggested model alterations. This greatly speeds up search. However, unless the desired solution is already present, this method makes the resulting CSP insoluble, which prevents us from finding link constraints with any of the above methods. On the other hand, since it is a very quick method for determining whether the desired solutions are present in the problem, it can be used in tandem with an all-solutions search, so that the latter can be avoided in these cases.

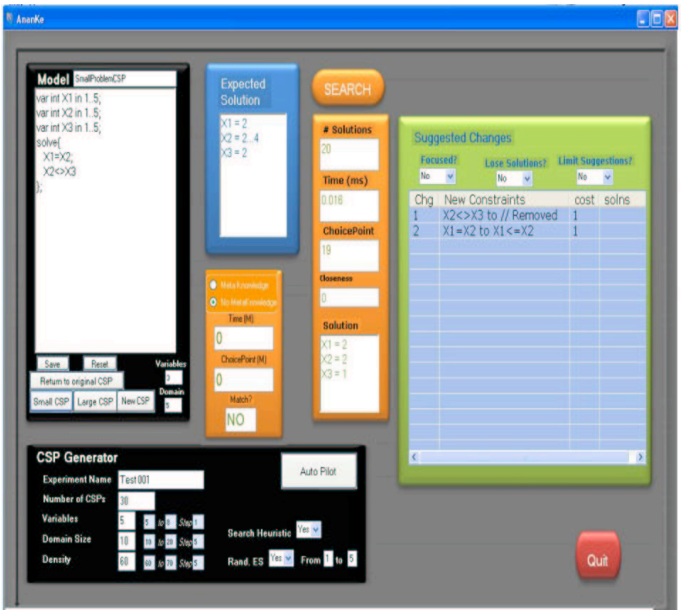


Figure 1. Ananke 1.0 Interface.

3 ANANKE 1.0

The ideas described above were implemented using Visual Studio. The code for this version of Ananke is written in Visual Basic. ILOG solver is used to solve particular CSPs. The present system runs under Windows XP on a Dell Precision PWS 390 (2.66 GHz, 2 GB RAM).

The basic user interface is shown in Figure 1. Different windows display the original or current CSP (black-bordered window on upper left), allow the user to enter an expected solution (topmost window to right of CSP window), show statistics pertaining to search (window below SEARCH button), and present suggestions in accordance with the limits placed on search by the user (large window on right-hand side). The DEBUG button, which initiates a search for suggestions, appears to the right of the SEARCH button when appropriate). There is also an area on the lower left for setting up and running experiments.

In the example shown in this figure, a small problem is displayed having three variables and two constraints. The user has specified a solution where X1=2, X3=2, and X2 should have a value in the range 2 ... 4. The system has found a best-match with X1=2, X2=2 and

X3=1, i.e. with two variables matching the specified assignments. A subsequent search (not restricted to link constraints) has discovered two changes to the original problem that will allow one of the desired solutions: one in which the inequality constraint between X2 and X3 is removed, and one in which the equality constraint between X1 and X2 is changed to a less-than-or-equals constraint.

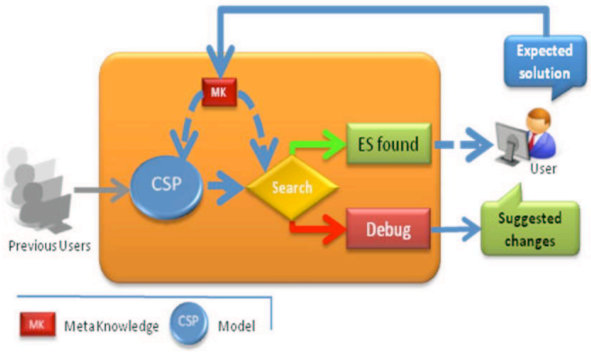


Figure 2. Ananke Debugging Overview.

Figure 2 gives an overview of the basic debugging procedure. After the current model of the problem is loaded, the user enters an expected solution (which may be a complete or partial solution and may also include ranges for some or all of the variables). This becomes part of the metaknowledge. The system then conducts an all-solutions search on the CSP represented by the model. Throughout this search, it records the maximum number of assignments in a solution that match the expected solution; this is the "closeness" of that solution, and the maximum closeness is presented at the end of the search along with a solution that gives that closeness. (The present version of Ananke does not show alternative solutions with maximum closeness, which can involve different subsets of variables. This may be included in latter versions. In any case, it does not affect the subsequent search for constraint changes that allow the expected solution to be incorporated into the model.) If the closeness is less than 100%, the debugging process is initiated.

Debugging search uses the CSP induced by the ms-constraints (which simply means that the suggested assignments are added to the model in the form of constraints) while carrying out a meta-search through the space of possible CSPs. This is done with a branch-and-bound algorithm that uses a specified cost-function to find minimal-cost changes to the present set of constraints (described in detail in the next section). (In Ananke 1.0 the cost function is chosen by the programmer, but in later versions it will be possible to do this through the interface. The coding required to do this is simple; the problem will be to present these choices to the user in a way that makes sense.)

The set of changes may include the entire set of best-cost solutions, or if the user has limited the search by indicating that only link constraints are to be tested, then suggested changes will only involve those constraints. As an example, the first suggestion shown in Figure 1 is obtained if search is restricted to the link constraint, X2 $\neq$ X3; while unrestricted search produces the two suggestions shown. In this case, the suggestion obtained under unrestricted search preserves both constraints, while that restricted to the link constraints does not.

4 THE DEBUGGING ALGORITHM

This section describes the branch and bound algorithm that is the core of the debugging process. The solutions sought in this process are sets of constraint changes that give the best cost according to some metric. The number of constraints in such a set can vary from 1 to n. However, if we allow the null change as one of the possible changes in a constraint, then all constraints will be involved in every solution; obviously, we can obtain the solution we want by discarding those constraints with null changes after search is over. This allows us to treat the metaproblem as an ordinary CSP whose variables are the constraints potentially subject to change, and whose values are the allowable changes.

Pseudocode for the branch and algorithm is shown in Figure 3, where the algorithm is presented in iterative form (its form in the present implementation).

The algorithm is also enhanced by preliminary identification of connected components. For disconnected graphs, if there are connected components that do not include any unassigned variables, the constraints in these components can be disregarded during search. This has no effect on the correctness of the algorithm, but it can enhance its efficiency under these circumstances.

```
meta-branch-and-bound()

currentCost = 0
bestCost = ∞
searchLevel = 0

while searchLevel ≥ 0
    if searchLevel > n                    'new solution found
        create and save newCSP based on present metamodel
            and ms-constraints
        if newCSP has a (CSP) solution
            if currentCost < bestCost
                set bestSolutions to solution
                set bestCost to currentCost
            else
                add solution to set of bestSolutions
        decrement searchLevel

    else if nextRelaxation ≤ maxRelaxation at searchLevel
        if currentCost + cost of nextRelaxation > bestCost
            reset data structures at this level
            decrement searchLevel
        else
            add nextRelaxation to partialSolution
            save nextRelaxation and update currentCost
            increment searchLevel
    else
        reset data structures at searchLevel
        decrement searchLevel
```

Figure 3. Pseudocode for branch and bound algorithm for search for the set of best-cost solutions in a space of metamodels using a monotonic cost function.

5 EXPERIMENTAL RESULTS

Our first experiments used a familiar metamodel whose constraints are based on the standard relational operators, $=$, $>$, $<$, $\geq$, $\leq$, and $\neq$, which can be arranged into a lattice structure consistent with the number of viable tuples. The cost function was that noted in the subsection on Metrics, where each step upward in the lattice incurs a cost of 1.

Figure 4 shows some representative results that demonstrates the expected improvement in search efficiency with ms-constraints of increasing arity (i.e. increasing numbers of ms-variables, here shown in proportion to the number of variables in the original problem). For all but the smallest partial solutions, the savings is over 90%.

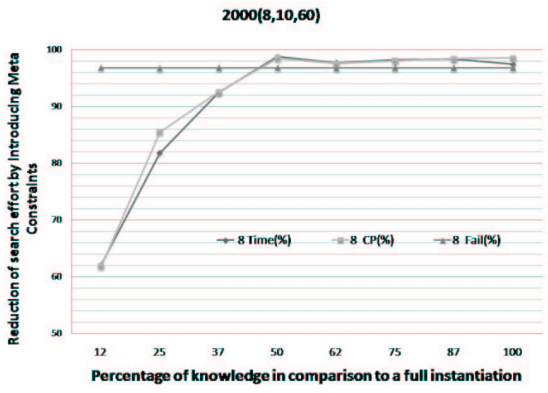


Figure 4. Search improvement with inclusion of ms-variables.

In other experiments, we compared suggestions under focused and unfocused conditions, i.e. using or not using linked constraints. In these experiments, 50 problems were generated per condition, and the size of the missing solution was 0.8 of the number of variables (rounded to the nearest integer). No attempt was made to control for the size of the best match, except to discard occasional cases where this was 100%.

In each case, more suggestions (i.e. sets of relaxations that allow the ground solutions required by the user) are found with unfocused search, while at the same time, there are on average fewer changes (i.e. constraint relaxations) per suggestion. For example, for <5,5,0.40> problems there were 1.2 suggestions per problem in the focused case and 1.6 with non-focused search, while the number of changes per suggestion were 2.2 and 1.5, respectively. For <8,5,0.25> problems there was 1 suggestion per problem with focused search and 1.4 with non-focused, while the number of changes per suggestion were 2.6 and 2.2. In these cases, based on a simple partial ordering of constraint relaxations, the cost of a suggestion is the same as the number of changes.

6 RELATED WORK

The present work is related to a small but growing body of work on constraint acquisition. This work involves procedures derived from psychology and machine learnig for acquiring discrete concepts based on examples; in this case, the examples are solutions and non-solutions. As a result, much effort has been made on developing efficient sets of queries that will enable a CSP model to be obtained [8] [2]. In the present work, we are attempting to adjust an existing model. Thus, while work in constraint acquisition attempts to impose restrictions on a space of possible models, the intention of the present work is to find models that are minimally distant from an existing model. As a result, the techniques employed in these respective tasks are quite different. Nonethess, a natural extension of the present work would be to combine the present approach with concept learning, where the system not only finds a set of minimally distant solutions but then presents examples in order to distinguish among them (or even larger sets).

In the present debugging task, the distinction between complete and partial solutions is not critical, although our algorithms are naturally more efficient when the solution set is more restricted. Although this may not be a critical difference for constraint acquisition either, it is something that has not to our knowledge been considered in this context. Finally, we should note that when the set of possible constraint relaxations is extended, e.g. if $X > Y$ can be replaced by $X > Y - k$ instead of by another relational operator, this will not have a marked effect on task difficulty although the version space is greatly extended with consequent difficulties for pure acquisition.

Another related area concerns "explanations" for assignment failures (e.g. [6] [5] [1] [7]). This work considers a similar type of problem, but from the other 'end of the telescope', in that the goal is to find adjustments to an invalid complete or partial solution that will satisfy the constraints that disallow it. In other words, the emphasis in explanation research is on finding correct solutions, while our concern is properly defining the problem. For this reason, it is not necessary to move to the level of metaknowledge to obtain explanations. However, explanations in the form of minimal conflict sets can serve as another heuristic method for guiding the debugging process, so there is much here that is of potential value for the present research.

Some of our heuristic methods resemble methods and issues in the field of dynamic CSPs. In particular, the focus heuristics we have considered are similar in intent to the concerns, e.g. for solution stability, that have inspired algorithms such as local changes [9]. At the same time, there is again a fundamental difference in intent, since our basic concern is finding a model that is consistent with a solution, rather than finding a solution consistent with a given model.

The context established here of searching for alternative models in a space of possibilities with a distance metric is reminiscent of the partial CSP (PCSP) framework for overconstrained problem solving [3]. In fact, in that earlier work a search through a space of relaxed CSPs was considered but that approach was not pursued. Since the concern in the PCSP context was to find an existing assignment with optimal properties (e.g. minimal number of violated constraints), the algorithms were different in character from the one presented here. Again, however, there is a potential for interplay between that field and the present work.

7 CONCLUSIONS AND FUTURE WORK

In this paper we have presented a new approach to constraint model debugging, and we have demonstrated the viability of these ideas with a working implementation. There are many extensions to consider. Perhaps the most obvious is to extend the system to handle a broader class of constraints and relaxations. More importantly, we need to consider the source(s) of metaknowledge in greater depth. That is, how do we decide what the alternatives for a given constraint should be?

We have also spoken about possible metrics to use in this context, as well as heuristics. Ideally an interface would allow users to choose different measures of solution quality, and then automatically choose appropriate heuristics to efficiently find solutions according to the chosen metrics.

Although the present system only makes suggestions that are monotonic with respect to allowed solutions, we may also wish to

be able to operate in a non-monotonic context. The changes we need or wish to make may remove some solutions at the same time as they add others. Since the system is assisting the user in searching the space of possible models, it needs to help the user avoid 'infinite loops'. On the other hand, users may want the right to change their minds, and e.g. remove solution s to admit solution t. Ultimately, we would like to be able to provide the user with a choice as to how 'paternal' the system should be, e.g. the user might say "never make a suggestions that would cause the removal of a solution I have added" or "only make such a suggestion as a last resort" or "tell me whenever you are making such a suggestion".

We also think that this approach can be extended to a context in which we are not merely detecting 'bugs' that prevent our model from being complete or correct, but correcting them in order to improve our model: where debugging becomes more of a discovery process. We illustrate with a simple example: Suppose we were designing a device such as a teapot that is heated through an electrical connection in a base. And suppose we were working with a CAD tool that allowed one to manipulate a mock-up of this device. In doing this, we find that any rotation around the vertical axis requires one to move the base as well as the teapot itself, but it would be better if the teapot could be rotated independently. However, this possibility, this 'solution', is not allowed by the present model.

We believe that using metaknowledge supplied by human-computer interaction to guide search through a metaspace of models provides a powerful approach to the CSP acquisition challenge. The broader vision for the future of Ananke is an environment that can provide constraint knowledge acquisition and maintenance assistance in a variety of forms to take best advantage of the user's experience while moving as much of the burden of the process as possible from human to machine.

REFERENCES

[1] J. Amilhastre, H. Fargier, and P. Marquis, 'Consistency restoration and explanations in dynamic csps - application to configuration', *Artificial Intelligence*, **135**, 199–234, (2002).

[2] C. Bessière, R. Coletta, B. O'Sullivan, and M. Paulin, 'Query-driven constraint acquisition', in *Proc. Twentieth Internatioal Joint Conference on Artificial Intelligence-IJCAI'07*, pp. 50–55, (2007).

[3] E. C. Freuder and R. J. Wallace, 'Partial constraint satisfaction', *Artificial Intelligence*, **58**, 21–70, (1992).

[4] E. C. Freuder and R. J. Wallace, 'Suggestion strategies for constraint-based matchmaker agents', *International Journal on Artificial Intelligence Tools*, **11**(1), 3–18, (2002).

[5] U. Junker, 'Quickxplain: Conflict detection for arbitrary constraint propagation algorithms', in *IJCAI 2001 Workshop on Modelling and Solving Problems with Constraints*, (2001).

[6] N. Jussien and V. Barichard, 'The paLM system: Explanation-based constraint programming', in *Proc. CP 2000 TRICS Workshop*, pp. 118–133, (2000).

[7] B. O'Callaghan, B. O'Sullivan, and E. C. Freuder, 'Generating corrective explanations for interactive constraint satisfaction', in *Principles and Practice of Constraint Programming -CP2005. LNCS. No. 3709*, pp. 445–459, (2005).

[8] S. O'Connell, B. O'Sullivan, and E. C. Freuder, 'A study of query generation strategies for interactive constraint acquisition', in *Applications and Science in Soft Computing*, pp. 225–232, (2003).

[9] G. Verfaillie and T. Schiex, 'Solution reuse in dynamic constraint satisfaction problems', in *Proc. Twelth National Conference on Artificial Intelligence-AAAI'94*, pp. 307–312, (1994).

Common Subexpression Elimination in Automated Constraint Modelling

Ian P. Gent and **Ian Miguel** and **Andrea Rendl**[1]

Abstract. Typically, there are many alternative models of a given problem as a constraint satisfaction problem, and formulating an effective model requires a great deal of expertise. To reduce this bottleneck, automated constraint modelling systems allow the abstract specification of a problem, which can then be refined automatically to a solver-independent modelling language. The final step is to *tailor* the model to a particular constraint solver. We show that we can eliminate common subexpressions in the tailoring step, as compilers do when compiling source code. We show that common subexpression elimination has two key benefits. First, it can lead to a dramatic reduction in the size of a constraint problem, to the extent that solving time is reduced by an order of magnitude when the number of nodes searched is the same. Second, it can lead to enhanced propagation and reduced search. The effect of this can be even more dramatic, leading to reductions in nodes searched and time taken by several orders of magnitude. Where the technique does not lead to improved search, we have not seen it cause a significant overhead. Therefore, we propose that common subexpression elimination is an important technique for constraint programming.

1 INTRODUCTION

Constraint solving of a combinatorial problem, such as timetabling or planning, proceeds in two phases. First, the problem is *modelled* as a set of decision variables and constraints that a solution must satisfy. The second phase consists of using a constraint solver to search for solutions to the model: assignments of values to decision variables satisfying all constraints. Modelling a large, complex problem using constraints does, however, require expert knowledge. Such experts are few in number, preventing widespread access to constraint solving. One important obstacle is the *modelling bottleneck*. Not only are there many possible models for a given problem, but the model chosen has a substantial effect on the efficiency of constraint solving, and selecting an effective model is difficult.

Recent work has addressed this problem by allowing the user to describe a problem at a high level in an abstract constraint specification language, such as ESRA [6], ESSENCE [7], or Zinc [3], *without* being forced to make detailed modelling decisions. An automated system, such as CONJURE [8] or Cadmium [18], transforms this specification into a concrete model. This step is similar to program compilation. A compiler and a modelling system both refine a high-level language to an intermediate representation that is flattened to a target machine. The difference lies in the processed data: compilers deal with a set of instructions, Constraint Modelling deals with a set of relations.

[1] University of St Andrews, UK, email: {ipg,ianm,andrea}@cs.st-and.ac.uk

Eliminating common subexpressions is a technique that has been used successfully in Compiler Construction [5]. The idea is to enhance a program by detecting common pieces of code: if two pieces are equivalent, one piece can be omitted. Hence a reduction in execution time and memory usage is achieved.

This paper shows that eliminating common subexpressions in the context of constraint modelling conveys two key benefits. First, it can produce a significant reduction in the size of a constraint model. Second, it can lead to improved constraint *propagation* (inferences made by the constraint solver), and therefore dramatically reduced search. We show experimentally that the first benefit can lead to an order of magnitude improvement in run time in a constraint solver, while the second benefit can give several orders of magnitude improvement.

2 BACKGROUND

A constraint model is defined by a finite set of decision variables and a finite set of constraints on those variables. A decision variable represents a choice that must be made in order to solve the problem. The finite *domain* of potential values associated with each decision variable corresponds to the various options for that choice. A good model may be solved quickly while a bad model might not be solvable in a practical amount of time. An efficient model exploits both the modelling language's features and the constraint solver's strengths. Hence choosing an efficient representation requires a lot of expertise. Automated modelling seeks to ease this burden by automating as much as possible of the modelling process.

2.1 Automated Constraint Modelling

A number of approaches have been taken to automated constraint modelling. For example, the CONACQ [4] system uses machine learning to formulate a model from a set of solutions and non-solutions provided by the user. The O'CASEY system [15] uses case-based reasoning to store, retrieve and reuse constraint programming experience. In this paper our focus is on automated modelling through *refinement* of an abstract specification. In particular, we will discuss our approach in the context of the ESSENCE / CONJURE system, but it is equally applicable to similar systems such as Zinc / Cadmium. Indeed, common subexpression elimination could be used as a post-processing step to improve an existing model.

The ESSENCE language allows the specification of a problem *abstractly*, i.e. without making modelling decisions. ESSENCE, like Zinc, provides decision variables whose domain elements are combinatorial objects, such as sets, functions, or relations. Furthermore, these objects can be nested so that an individual variable may represent a set of sets, a set of sets of relations, and so on. This specifica-

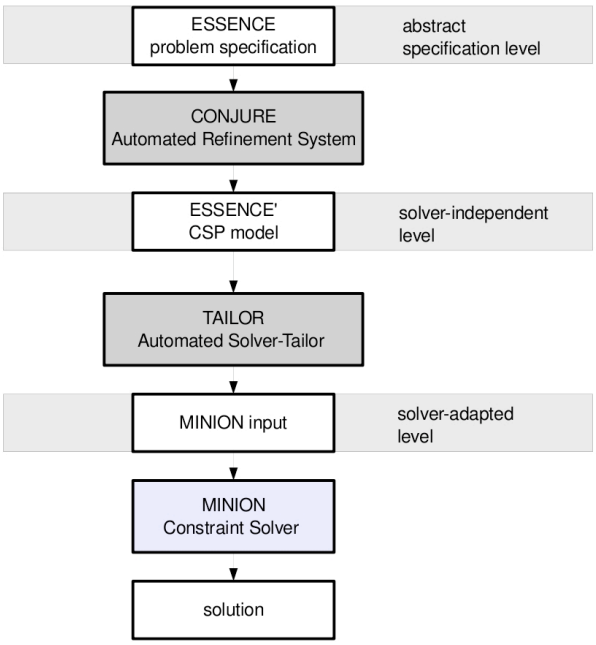


Figure 1. Automated Constraint Modelling with ESSENCE and CONJURE.

tion is refined automatically by the CONJURE system to a solver-independent constraint modelling language ESSENCE′. ESSENCE′ is a version of ESSENCE that has abstraction removed (principally, domain values are atomic) and provides facilities common to existing constraint solvers and toolkits. An ESSENCE′ model is adapted, or 'tailored' to a particular constraint solver using the TAILOR system [12]. The tailoring task consists of mapping ESSENCE′ expressions to the set of constraints provided by the target solver, as explained below. In this paper we will use MINION [11] as our target solver. An overview of the automated modelling process is given in Figure 1. At present, we are eliminating common subexpressions in the tailoring stage. In future, we plan to lift this work to the refinement stage.

3 TAILORING CONSTRAINT INSTANCES TO SOLVERS

Our approach focusses on tailoring problem instances rather than problem classes to a target solver. An instance is obtained from a class by giving a value for each parameter in the model (e.g. by giving $n = 8$, we obtain the 8-queens instance). Hence, each occurrence of a parameter in the model is instantiated to its associated value, which promotes tailoring steps such as evaluation and flattening. In our implementation, TAILOR adapts solver-independent constraint models to a particular solver by the following steps:

- Insert parameter values to obtain an individual instance
- Normalise the problem instance
- Flatten ESSENCE′ constraints (and variable datastructures) to conform those provided by the target solver
- Map flat, normalised instance to the target solver

Some constraint solvers, such as Minion, require individual instances as input. For others, our conjecture is that tailoring instances is worthwhile because of the extra information provided by instantiating the parameters. In the following we discuss the tailoring steps that are crucial for detecting common subexpressions efficiently.

3.1 Normalisation: A Prelude to Common Subexpression Detection

Our normalisation of ESSENCE′ has two components: evaluation and ordering. These are applied in an interleaved manner until a fixpoint is reached (the normal form). They are described below. We normalise expressions not only after parsing but also during flattening, when we unroll more complex expressions, such as quantifications. We do not apply any kind of factorisation of expressions.

3.1.1 Evaluation

Evaluation is particularly powerful when tailoring a problem instance to a target solver. Care is necessary in deciding the extent of evaluation: in some cases further evaluation might improve an instance but increase tailoring time and hence impair the (combined) modelling and solving process. Therefore, the expression evaluation included in our normalisation is simple, and cheap to perform. We evaluate constant expressions and apply several simple algebraic transformations, such as algebraic identity or algebraic inverses. We give some examples below.

$3 * 4 - 2$	$\longrightarrow$	10	Constant Evaluation
$exp + 0$	$\longrightarrow$	exp	Algebraic Identity
$exp - exp$	$\longrightarrow$	0	Algebraic Inverse
$exp \wedge false$	$\longrightarrow$	*false*	Logic

3.1.2 Ordering

We define a total order $\leq_o$ over the expressions of ESSENCE′. An ESSENCE′ model is transformed into a minimal form with respect to this order. The order represents a hierarchy of expressions, based on their complexity: Expressions on the bottom of the hierachy are expensive ones, such as non-linear expressions. Constants are at the top of this order, followed by variables and arrays. Further down the order are constraint types such as equalities, disequalities, inequalities, and special constraints like 'all-different'. Linear expressions come before non-linear expressions.

$$a + b + c + d \quad \leq_o \quad a * b$$

Expressions of different type are ordered based on their position in the order. Expressions of the same type are ordered recursively; each type has self-comparison rule. An equality constraint, for example, is ordered by examining the left argument first, followed by the right. The base case is where two constants or two variables are compared. In the former case, the comparison is by value, with least first. In the latter, the comparison is by name and domain. To illustrate, consider the following normalisations:

$$\begin{array}{lcl} x_4 + x_3 \neq x_2 + x_1 & \longrightarrow & x_1 + x_2 \neq x_3 + x_4 \\ x_5 * x_6 = x_8 + x_7 & \longrightarrow & x_7 + x_8 = x_5 * x_6 \end{array}$$

3.2 Flattening

It is common for constraint languages to support complex constraint expressions by re-writing, or *flattening*, them into a conjunction of

simple constraints. In general, this mechanism is straightforward: replace a complex subexpression by an auxiliary variable that represents the subexpression. For example, an arithmetic expression, such as $a * (b + c)$, is flattened by replacing $b + c$ by an integer auxiliary variable aux_i and introducing an additional constraint $aux_i = b + c$. A relational expression, such as $a \Rightarrow (b \wedge c)$, is flattened by replacing $b \wedge c$ with a Boolean auxiliary variable aux_b. The equivalence between aux_b and $b \wedge c$ is expressed by a so-called *reification* constraint *reify*$(b \wedge c, aux_b)$ that corresponds to $aux_b \Leftrightarrow b \wedge c$ (aux_b is true if and only if the reified constraint $b \wedge c$ is satisfied).

	Unflattened	Flattened
Arithmetic Expression	$a * (b + c)$	$aux_i = a + b$ $a * aux_i$
Relational Expression	$a \Rightarrow (b \wedge c)$	*reify*$(b \wedge c, aux_b)$ $a \Rightarrow aux_b$

More complex expressions require several flattening steps to flatten completely, each introducing another auxiliary variable and equality constraint or reified constraint, respectively. As an example of a more complex expression, consider the constraint describing the 'legal moves' in the action-based model of the Peg Solitaire problem from [14] in Table 1.

4 COMMON SUBEXPRESSION ELIMINATION

This section discusses different sources of common subexpressions, together with efficient ways of detecting and exploiting them.

We say that two expressions are *common* (or equivalent) if they take the same value under all possible satisfying assignments. We distinguish between two types of equivalent subexpressions: subexpressions that are *syntactically* equivalent and subexpressions that are *semantically* equivalent. Syntactically equivalent expressions are *written* in the same way, such as a pair of occurrences of $a * b$. Semantically equivalent expressions *mean* the same thing, which can be deduced by their operational semantics, such as from the equivalence relation $a * b = c$. Clearly, syntactically equivalent expressions are also semantically equivalent.

Due to properties such as commutativity, many semantically equivalent logical and arithmetic expressions can be written so as to be syntactically distinct. A simple example is $a + b$ versus $b + a$. By normalising a constraint model (as we describe in Section 3.1) prior to common subexpression detection, the test for semantic equivalence is, in many cases, therefore reduced to the much cheaper test for syntactic equivalence.

The process of common subexpression elimination is straightforward. We record every two expressions that we denote equivalent. According to our expression ordering (see Section 3.1.2) the smaller expression is stored in a hashmap as representative for the greater expression. Whenever the greater expression re-occurrs in the constraint model, it is replaced by the smaller expression. Since the ordering captures the complexity of expressions, a subexpression is always replaced by a more effective subexpression.

Most common subexpressions arise in quantified expressions that are unrolled during flattening. Although a constraints expert can, of course, recognise common subexpressions and perform elimination manually, it is likely that a non-expert would not. Furthermore, even for an expert, performing this step in a complex model can be laborious and, without care, a source of error.

4.1 Explicit Common Subexpressions

A very simple example of common subexpressions is when the model contains a constraint of the form $X = Y$. We shall refer to constraints of this form as *explicit* equivalences, since they are given directly in the model.

4.1.1 Equivalence between Atomic Expressions

The simplest case of explicit common subexpressions $x = y$ is where x and y are atomic expressions, i.e. variables or constant values. The standard enhancement approach is to use x for every occurrence of y and, if y is a variable, to remove y from the set of variables, thus saving a variable. This approach has been extensively studied [1, 2, 16].

4.1.2 Equivalence between Compound Expressions

The general case of explicit common subexpressions $X = Y$ occurs when X and Y are arbitraily complex expressions. As example, consider the expression $x = y * z$. According to our ordering, x is cheaper than $y * z$ and we can replace every further occurrence of $y * z$ with x. Though exploiting equivalence between compound expressions can yield very effective results, this case mostly occurs in models formulated by non-experts.

4.2 Common Subexpressions Introduced During Flattening

The flattening process, which was explained in Section 3.2, naturally introduces a large number of equalities and reification constraints, which are a rich source of common subexpressions. If a certain subexpression appears again, we can simply *re-use* the auxiliary variable that already represents the subexpression, as the simple example below demonstrates:

Unflattened	Standard Flattening	Enhanced Flattening
$a + x * y = 0$	$aux_1 = x * y$	$aux_1 = x * y$
$x * y + b = t$	$a + aux_1 = 0$	$a + aux_1 = 0$
	$aux_2 = x * y$	$aux_1 + b = t$
	$aux_2 + b = t$	

In our implementation, we flatten expressions bottom-up, i.e. expressions are flattened starting from the leaves of the expression tree. We maintain a hashmap that maps all previously flattened subexpressions to their corresponding auxiliary variables. Whenever we flatten a new subexpression we look up the hashmap for an equivalent expression: if we find an equivalent expression, we replace it with the respective auxiliary variable. This approach reduces the time required to match subexpressions and the memory we spend to collect previously flattened subexpressions. Note the importance of normalisation here: it is much easier to detect equivalence of normalised subexpressions.

The benefits we gain are great. First, if an instance contains common subexpressions of this kind, we save a variable and a set of constraints (depending on the complexity of the common subexpression) for every subexpression. Below, we report results to show that this effect on its own can reduce solving time by an order of magnitude, even without reducing the numbers of nodes searched.

We obtain a second large benefit, with even greater potential. This is that we can get additional propagation through re-using auxiliary variables. To see how this could happen, consider again the example

above. Suppose that the domains of x and y are both $\{1, 2\}$. The domain of a is therefore $\{-4, -2, -1\}$ because $a + x * y = 0$. During search, we might set $b = 0, t = 2$. From this we can deduce $x * y = 2$ and in the standard flattening we get $aux_2 = 2$. However, we can deduce nothing about x or y because either $x = 1, y = 2$ or $x = 2, y = 1$ is possible, so nothing propagates through to aux_1 or a. When we use enhanced flattening, we share the same variable, so we deduce $aux_1 = 2$ and immediately propagate to set $a = -2$. Of course this can propagate further, depending on the problem. Thus, the simple detection of common subexpressions can lead to reduced search. Not only can it do this in principle, we will see below that it can reduce search by a factor of more than 2,000 in practice.

Some solvers flatten their input themselves, such as the Eclipse Constraint Programming System [10] which does not eliminate common subexpressions [19]. However, most solvers, such as MINION or Gecode [9] take a flattened model as input, hence flattening (in combination with common subexpression elimination) has to be done by the modeller - a tedious task, even for an expert (consider eliminating all common subexpressions of the 'legal moves'-constraint of the Peg Solitaire model in Table 1!). Therefore both Constraint novices and experts benefit from automated common subexpression elimination: poor models are drastically improved and good models might be improved if they contain common subexpressions without increasing tailoring time significantly.

Peg Solitaire **Action model description**
We represent the board by *bState*, a list of squares for each step of the game. Every possible peq-move is assigned to a number between 1 and 76 and the array of variables *moves* holds the corresponding move for each step.

```
 0 given    noSteps : int
 1 letting  transitionStep :
 2            matrix indexed by [int(1..76),int(1..3)] of int(1..33) be ...
 3 letting  transitionNumber :
 4            matrix indexed by [int(1..33),int(1..33)] of int(0..76) be ...
 5 letting  STEPS be domain int(0..noSteps)
 6 letting  FIELDS be domain int(1..33)
 7
 8 find     bState : matrix indexed by [ STEPS, FIELDS ] of bool
 9 find     moves : matrix indexed by [int(0..noSteps-1) ] of int(1..76)
10
11 such that
12   . . .
13   $ legal moves
14   forall step : int(0..noSteps-1) .
15     forall f1,f2 : FIELDS .
16
17       $ if there exists a legal move from f1 to f2
18       (transitionNumber[f1,f2] != 0) ⇒
19
20       $ and we make that transition, the following holds..
21       ( (moves[step] = transitionNumber[f1,f2]) ⇔
22
23         ( (bState[step, f1] > bState[step+1,f1]) /\
24
25           (bState[step,transitionStep[transitionNumber[f1,f2],2]] >
26            bState[step+1, transitionStep[transitionNumber[f1,f2],2]]) /\
27
28           (bState[step, f2] < bState[step+1, f2]) /\
29
30           forall field : FIELDS .
31             ( (field != f1) /\
32               (field != transitionStep[transitionNumber[f1,f2],2]) /\
33               (field != f2)
34             ) ⇒
35               (bState[step,field] = bState[step+1,field])
36           )
37       )
```

Table 1. Segment of the Peg Solitaire *Action* model [14] formulated in modelling language ESSENCE′. A summary of the model is given in Table 2

4.2.1 Example: Common Subexpressions in Peg Solitaire

As an example for common subexpressions, consider the partial model of the Peg Solitaire Problem [14] in Table 1. Peg Solitaire is a gamed played on a board with holes and pegs to arrange. The aim in the standard version of the game is to perform checkers-like moves to remove all pegs but one from the board.

We represent the 33 fields (holes) on the board by booleans: *true* states that a peg is in the hole and *false* states that the hole is empty. The board changes after every move, so we represent the board states by the matrix *bState*, where the ith vector represents the field-variables for the ith step in the game. There are 76 possible moves on the board and the 1-dimensional matrix *moves* holds the variables for each move made in the game. The constant matrix *transitionNumber*[f_1,f_2] gives the corresponding transition number (ranging from 1 to 76) when making a move from field f_1 to f_2. *transitionStep* [*step*, i] gives the field-variable that is involved when performing the step with number *step*.

We constrain the move chosen to be legal using a universal quantification in line 14. A summary of the 'legal moves'-constraint is given in Table 2. Recall that such quantified 'loops' must be unrolled for constraint solvers such as Minion or Gecode. As we do so, common subexpressions arise between the expressions obtained for different values of the quantified variable. An example is the inequality in line 23 that is nested in a conjunction,

$$bState[step, f1] > bState[step + 1, f1]$$

When the quantification is unrolled for *step*=0 and f_1=2, it yields the subexpression

$$bState[0, 2] > bState[1, 2]$$

that re-occurs every time field 2 is involved in another move at step 0. The same holds for the other inequalities from lines 25 and 28.

Summary of action-centric model of Peg Solitaire where *move* is an array of variables representing the moves required to solve the puzzle, *bState* is an array of variables representing the state of the board at each step, t ranges over the steps in the sequence of moves, m is a move, start(m), mid(m) and end(m) return the three positions affected by move m, and unchanged(m) returns the set of positions *not* affected by move m.

forall t in 1..31 .
forall m in 1..76 .
  move[t] = m ↔
    bState[$t-1$, start(m)] > *bState*[t, start(m)] ∧
    bState[$t-1$, mid(m)] > *bState*[t, mid(m)] ∧
    bState[$t-1$, end(m)] < *bState*[t, end(m)] ∧
    forall u in unchanged(m) . *bState*[$t-1$, u] = *bState*[t, u]

Table 2. Summary of action-centric model of Peg Solitaire

5 EXPERIMENTAL RESULTS

In this section we compare models that we tailor (as described in Section 3) either with or without common subexpression elimination. We present a selection of problems that we formulate in ESSENCE′ without applying symmetry breaking. Then we tailor the ESSENCE′ model to a MINION instance using the tool TAILOR, in which we have implemented (optional) common subexpression elimination. For each problem instance, we generate two different MINION input files: one that is tailored by eliminating common subexpressions and one that is not. Both models are solved on the same machine (Dual-core Intel P4 at 3GHz with 1.5Gb RAM) using MINION v0.5. We apply the same variable ordering heuristic (decision variables first,

n	Tailoring (s)		Solving Time (s)		Search Nodes		Common	Aux Variables		Constraints	
	♠	♡	♠	♡	♠	♡	Subexpr.	♠	♡	♠	♡
5	0.36	0.37	9.49	0.04	400,399	1870	1,230	1,440	200	1,486	256
6	0.41	0.42	1809.53	0.39	79,159,269	32,964	2,248	2,535	287	2,614	366
7	0.53	0.49	48,020.50	8.58	1,448,334,418	604,206	3,710	4,101	391	4,206	496

Table 3. Solving performance and model features of Peaceable Army of Queens models with (♡) and without(♠) common subexpression elimination

then auxiliary variables) and same value ordering heuristic (ascending) in both cases. We compare the models in several ways. As well as solving performance we report tailoring time. We also look at features of the tailored instances, such as the number of constraints and auxiliary variables.

5.1 Golomb Ruler

The Golomb Ruler problem is to find a ruler of minimal length with n ticks such that the distance between every two ticks is different. Our results with common subexpression elimination are very interesting: we take the basic model from [13] which uses quarternary constraints to express the distances between the ticks. Applying common subexpression elimination on the basic model automatically yields the enhanced distance model from [13]. Hence this example demonstrates how weak models can automatically be enhanced to advanced, effective models from the literature. The results are given in Table 4: we gain a great reduction in search time but also in search space.

n	Common Subexpr.	Solving Time ♠	Solving Time ♡	Search Nodes ♠	Search Nodes ♡
7	679	0.97	0.04	2,507	1,996
8	1260	26.80	0.35	22,508	17,427
9	2148	513.54	3.27	188,026	141,503
10	3435	>7,461.07	36.32	>1,406,328	1,114,964

Table 4. Solving performance of Golomb Ruler instances with(♡) and without(♠) common subexpression elimination

5.2 Peg Solitaire

We formulated two different models of Peg Solitaire. The first is taken from [14] and is *state*-centric: for each possible change to the state of the board, a constraint is added specifying the moves that might be responsible. We also experimented with a novel *action*-centric model: a constraint is added for each possible action, specifying the changed and unchanged parts of the board. The constraints are briefly summarised in Table 2.

In the action-centric model, we reduce the number of auxiliary variables from 87,172 to 6,603, and the number of constraints from 89,625 to just over or under 9,000 depending on the starting position. In the state-centric model, we reduce the number of auxiliary variables from 313,720 to 12,989 and the number of constraints from 316,886 to 16,155. We present solving results in the two models (from three different starting positions) in Table 5. In the action-centric model, we get no reduction in search nodes, a small increase in tailoring time, but an order of magnitude reduction in run time. In the state-centric model, we do in fact get a reduction in search of about a factor of 3, as well as a reduction in time taken per node. Note that the elimination of common subexpressions reverses the performance of models. That is, the action-centric model is best without subexpression elimination, but when it is used the state-centric model searches faster. When tailoring time is also taken into account, the action-centric model is fastest overall on the easiest instance, but state-centric remains best for the two harder instances.

start field	Tailoring (s) ♠	Tailoring (s) ♡	Solving (s) ♠	Solving (s) ♡	Search Nodes ♠	Search Nodes ♡
17	5.88	5.97	31.7	3.2	10,269	10,269
10	5.87	6.18	4376.2	456.7	1,486,641	1,486,641
5	5.97	6.04	>7200	3,920.4	11,398,210	11,398,210
17	47.06	44.37	42.4	2.7	10,269	3,944
10	46.55	44.24	6,383.2	247.4	1,486,641	539,374
5	46.64	44.70	>7200	2,151.8	>1,784,832	3,066,971

Table 5. Action-centric (top) and state-centric (bottom) Peg Solitaire models with (♡) and without (♠) common subexpression elimination

5.3 Peaceable Army of Queens

The peaceable army of n queens problem is to place two equally-sized armies of white and black queens on an $n \times n$ chessboard such that no queen can attack a queen of the other colour. We formulate the 'basic model' of Smith *et al* [17] without symmetry breaking constraints in ESSENCE′. We compare performance (to find an optimal solution and prove its optimality) and the models in Table 3. Common subexpression elimination has a more dramatic impact than in the previous experiment. Here, we see the number of search nodes reduced from 1.5 billion to less than a million at $n = 7$, a factor of more than 2,000. Solving time is reduced even more, by more than 5,000 times at $n = 7$. These dramatic improvements occur through improved propagation after we have eliminated common subexpressions. They allow reasoning to occur over parts of the model which are separated in the vanilla model. Comparison with results of [17] is inconclusive. Our results without common subexpression detection are much worse than reported there, while results with it are similar. We do not know if this is because some feature of our model which is different in detail to theirs, or Minion propagates the same model worse, or whether Smith *et al* may have eliminated subexpressions without detailing it. None of these explanations would invalidate our main point, that common subexpression elimination can, on its own, make a very bad model much better purely automatically.

5.4 Balanced Incomplete Block Design (BIBD)

BIBD is problem 28 in CSPLib [20]. We use the standard model from the literature, consisting of 0/1 variables, sums and scalar products. The model does not contain common subexpressions so we cannot improve the model during flattening. Still, we generate two MINION models from each instance: one where we try to eliminate common subexpressions (in vain) and one without. We don't give a model comparison since the generated models are identical, but investigate tailoring and solving time in Table 6. This comparison is very interesting: we observe that we do not suffer significantly from the attempt to eliminate common subexpressions, even though there are none. Translation times are no more than 30% higher when failing to find any common subexpressions. We generate an identical model and get identical search results in terms of nodes searched, with very similar search times. Fluctuations in search time are presumably just the difference between separate runs. From this experiment we draw

the conclusion that the attempt to eliminate common subexpressions - even in vain - does not significantly slow down the modelling and solving process.

b, v, r, k, λ	Tailoring (s)		Solving Time (s)		Search Nodes	
	♠	♡	♠	♡	♠	♡
7,7,3,3,1	0.28	0.24	0.01	0.01	21	21
140,7,60,3,20	0.51	0.59	0.43	0.44	17,235	17,235
210,7,90,3,30	0.68	0.82	2.61	2.63	67,040	67,040
280, 7,120,3,40	0.90	1.15	9.92	9.51	182,970	182,970
315,7,135,3,45	1.04	1.26	16.05	17.05	278,310	278,310
385, 7,165,3,55	1.29	1.64	44.17	44.30	574,365	574,365

Table 6. Solving performance of BIBD models with (♡) and without(♠) common subexpression elimination. No common subexpressions were found.

6 RELATED WORK

Le Provost and Wallace discuss derivation and elimination of common subexpression during propagation in [2] but restrict their discussion to explicit atomic subexpressions. Harvey and Stuckey eliminate explicit atomic and linear subexpressions in their work on improving linear constraint representations in [1]. Neither study addresses common subexpression elimination during flattening nor elimination of non-linear constraints, as we do in our work.

In their work on interval analysis, Schichl *et al* [21, 22] discuss common subexpression elimination in models of mathematical problems represented as directed acyclic graphs. These studies have much in common with our work, and further examine the issue of propagation over common subexpressions. However, they do not include logical expressions, such as quantification, which we have identified as one of the main sources of common subexpressions.

7 CONCLUSIONS

We have shown that common subexpression detection, common in compilers, can be applied successfully to constraint modelling. We have shown that this can be implemented effectively as part of the TAILOR system, which translates models from a solver-independent modelling language to a target constraint solver.

Our experimental results show three things. First, we can obtain an order of magnitude improvement in run time simply by reducing the number of variables and constraints, with no change in search space. Second, we can obtain additional propagation, resulting in orders of magnitude improvements in the search space and run time. Third, although these improvements are not always possible, we do not pay a significant penalty where we cannot find common subexpressions. Taken together, the huge benefits outweigh the low costs, and common subexpression elimination should be considered wherever possible.

It could be argued that tailoring a simple model without common subexpression detection is a straw man, because other models perform better. There may be other models which are inherently better, with or without common subexpression detection. It is also true that any model reduction achieved automatically can also be achieved manually. However, these potential criticisms do not address the true point of our work. First, if we can improve a poor model by a factor of thousands in run time, we may avoid the need to spend time thinking of a better model. The result may be much better in terms of time required of an expert constraint modeller. Second, while common subexpression detection *could* be done by humans, it is *not* done in practice. In fact, it would often be impractical. A modeller would have to study a model looking for repeated expressions, and then remodel manually while avoiding mistakes in doing so. It is preferable to automate this process. Hence both modelling expert and novice benefit from automated common subexpression elimination.

Acknowledgements Ian Miguel is supported by a UK Royal Academy of Engineering/EPSRC Research Fellowship. Andrea Rendl is supported by a DOC fFORTE scholarship from the Austrian Academy of Science and UK EPSRC grant EP/D030145/1. We thank Warwick Harvey and our anonymous reviewers for their helpful comments.

REFERENCES

[1] W. Harvey, P. Stuckey. Improving Linear Constraint Propagation by Changing Constraint Representation, *Constraints*, 7, pp172–203, 2003.
[2] T. Le Provost, M. Wallace Generalized Constraint Propagation over the CLP Scheme. J. Logic Programming, 16(3), 1993.
[3] M. de la Banda, K. Marriott, R. Rafeh, M. Wallace. The modelling language Zinc. *CP*, 700-705, 2006.
[4] C. Bessiere, R. Coletta, F. Koriche, B. O'Sullivan. Acquiring Constraint Networks using a SAT-based Version Space Algorithm. In *AAAI*,, pp 1565-1568, 2006.
[5] J. Cocke, Global common subexpression elimination, *SIGPLAN Not.*, 5:20–24, 1970.
[6] P. Flener, J. Pearson, M. Agren. Introducing ESRA, a relational language for modelling combinatorial problems. *LOPSTR '03: Revised Selected Papers*, LNCS 3018, 2004.
[7] A.M. Frisch, M. Grum, C. Jefferson, B. Martínez Hernández, and I. Miguel. The design of Essence: A constraint language for specifying combinatorial problems. *IJCAI*, pp80-87, 2007.
[8] A. Frisch, C. Jefferson, B. Martínez Hernández, and I. Miguel. The rules of constraint modelling. *IJCAI*, pp 109–116, 2005.
[9] Gecode: a Generic Constraint Development Environment http://www.gecode.org
[10] The ECLiPSe Constraint Programming System http://eclipse.crosscoreop.com/
[11] I.P. Gent, C. Jefferson, and I. Miguel. Minion: A fast scalable constraint solver. *ECAI*, pp 98–102, 2006.
[12] I.P. Gent, I. Miguel and A. Rendl. Tailoring Solver-independent Constraint Models: A Case Study with Essence′ and Minion In *SARA*, pp 184-199, 2007.
[13] B. M Smith, K. Stergiou and T. Walsh. Modelling the Golomb Ruler Problem School of Computing Research Report 1999.12, University of Leeds, June 1999.
[14] C. Jefferson, A. Miguel, I. Miguel, A. Tarim. Modelling and Solving English Peg Solitaire. In *Computers and Operations Research* 33(10), pages 2935-2959, 2006.
[15] J. Little, C. Gebruers, D. G. Bridge and E. C. Freuder, Using Case-Based Reasoning to Write Constraint Programs. In *CP*, pp 983, 2003.
[16] B.A. Nadel. Representation Selection for Constraint Satisfaction: A Case Study Using n Queens. *IEEE Expert* 5:16-23, 1990
[17] B.M. Smith, K.E. Petrie, and I.P. Gent. Models and symmetry breaking for peaceable armies of queens. In *Proceedings CPAIOR 04*, pages 271–286, 2004.
[18] P. J. Stuckey, M. de la Banda, M. Maher, K. Marriott, J. Slaney, Z. Somogyi, M. Wallace, T. Walsh. The G12 Project: Mapping Solver Independent Models to Efficient Solutions. *CP*, 13-16, 2005.
[19] Warwick Harvey Personal Communication, May 2008
[20] CSPlib: A Library for Constraint Problems http://www.csplib.org/
[21] H. Schichl and A. Neumaier, Interval Analysis on Directed Acyclic Graphs for Global Optimization Journal of Global Optimization 33/4 (2005), 541-562
[22] X.-H. Vu, H. Schichl and D. Sam-Haroud Using Directed Acyclic Graphs to Coordinate Propagation and Search for Numerical Constraint Satisfaction Problems In *ICTAI 2004*, 72-81

Global Preferential Consistency for the Topological Sorting-Based Maximal Spanning Tree Problem

Rémy-Robert Joseph[1]

Abstract. We introduce a new type of fully computable problems, for DSS dedicated to maximal spanning tree problems, based on deduction and choice: *preferential consistency* problems. To show its interest, we describe a new compact representation of preferences specific to spanning trees, identifying an efficient maximal spanning tree sub-problem. Next, we compare this problem with the Pareto-based multiobjective one. And at last, we propose an efficient algorithm solving the associated preferential consistency problem.

Keywords: Consistency enforcing, Interactive methods, Multiobjective combinatorial optimization, Preferences compact representation, Spanning tree.

1 INTRODUCTION

Given an undirected graph $G = (V, E)$ with V the vertices and E the edges, a **spanning tree** x of G is a connected and acyclic partial graph of G. x is then always composed with $|V| - 1$ edges. We denote by $S_{ST}(G)$ the spanning trees set of G. For short, we write: $e \in x$, with $e \in E$, to say: e is an edge of the spanning tree x. More generally, we will assimilate x to its edges set. The classical problem of maximum spanning tree ($\Leftrightarrow$ ST/Σu/OPT) is defined as follow:

ST/Σu/OPT: Given an undirected graph $G = (V, E)$ and a utility $u(e)$ associated with each edge $e \in E$, the result is a feasible spanning tree x of G, maximizing the sum of utilities of edges in x, if such a tree exists. Otherwise, the result is 'no'.

Several consistency problems have been recently investigated on spanning trees. On the one hand, we note the consistency problem associated with feasible spanning trees of a graph [25]. Other investigations pointed out consistency associated with weighted spanning trees [8], and maximum spanning tree [9]. On the other hand, numerous local consistency problems combining classical spanning tree problems with other constraints have been investigated. For example, the diameter constrained minimum spanning tree problem (DCMST) [16].

Within non-conventional preferences, the situation is radically different. Very few consistency spanning tree problems have been investigated in literature. We cite a local consistency problem processed for the robust spanning tree problem with interval data (RSTID) [1].

Yet, the most of combinatorial problems from the real practical world require the modeling of imprecision or uncertainty, multiple divergent viewpoints and conflicts management, to wholly assess the solutions and to identify the best compromise ones. These singularities require more complex modeling of preferences [27, 21]. For now some decades, the OR/CP community scrutinizes combinatorial problems enabling non-conventional global preferences. Thus, we attended to the flowering of a great number of publications dealing with multiobjective combinatorial optimization problems (see [10, 3] for surveys). Nevertheless, a very few articles dealt with combinatorial problems with purely ordinal and/or intransitive preferential information. We mention the recent investigations in the scope of (i) decision theory with maximal spanning trees and maximal paths in a digraph [18], (ii) game theory with stable matchings (see [20] for a survey), (iii) algebraic combinatorial optimization [28, 5], or (iv) artificial intelligence with some configuration problems [4, 14] and with heuristic search algorithms [17, 14]. We decide to bring another stone to this building, with the concept of preferential consistency applied to the topological sorting-based maximal spanning trees problem.

The decision problematic of finding a suitable preferred solution is semi-structured: in the general case (beyond total preorders), a preferred solution fitted to the decision-maker cannot be only identified from the implemented preferential information. Preferred solutions are not all equivalent, some are partially comparable others are incomparable, and sometimes, there exists no optimal or maximal solution [27, 21]. To investigate these semi-computable problems, we will use the concept of Decision Support System (DSS) to explore the preferred solutions set. This exploration can be achieved other than by building iteratively new preferred solutions – as usually in multicriteria optimization –; For example, by describing this preferred set with the set of values present in at least one preferred solution. The notion of consistency, defined in Constraint Programming, gathers the theoretical surrounding of this descriptive approach of implicit sets. This is a reactive [26] and deductive approach of solving; In a polynomial number of actions (removings, instantiations and backtrackings), the user leads to a preferred solution.

Consequently, after an introduction on preference relations (§ 2.1), we make a brief presentation on compact representation of preferences (§ 2.2). We next point out a generalization of the maximum spanning trees problem: the maximal spanning trees problem (§ 3.1). So, we introduce (§ 3.2) preferential consistency, i.e. a template redefining consistency in order to take into account of peculiarities of combinatorial problems exploiting non-conventional preferences, followed by its using on the maximal spanning

[1] Université des Antilles et de la Guyane / Institut d'Etudes Supérieures de Guyane, French Guyana, France, e-mail : remy.joseph@caramail.com

trees. In general, most of relevant computable problems supporting the initial decision problem are intractable. Accordingly, we point out an easy suitable maximal spanning trees sub-problem (§ 4), based on a compact preference representation inspired by topological sorting (§ 4.1). In § 4.2, we give an example of using in the multicriteria context and we compare this sub-problem with the Pareto-based multiobjective version. Next, we design a global preferential consistency algorithm (§ 5) dedicated to it. We conclude (§ 6) with some perspectives.

2 PREREQUISITES IN DECISION THEORY

Throughout this article, we take place at a very general abstraction level, where global preferences are represented by a non complete, intransitive and even cyclic binary relation on the solutions space, but enabling a maximal set (there exist no solution strictly preferred to any of them). Here are some definitions:

2.1 Preference relation

Given a non-empty finite set S, a **(crisp binary) preference relation** [23, 27, 21] $\succcurlyeq$ of an individual on S is a reflexive binary relation on S ($\Leftrightarrow \succcurlyeq \subseteq S \times S$ and $\forall\, x \in S, (x, x) \in \succcurlyeq$) translating some judgments of this individual concerning his preferences between the alternative elements of S. For every couple of elements x and y of S, the assertion « $x \succcurlyeq y$ » is equivalent to « $(x, y) \in \succcurlyeq$ » and means that « x is at least as good quality as y for considered individual ». A preference relation $\succcurlyeq$ carries out a partition of $S \times S$ into four fundamental relations:

(**indifference**) $x \simeq y \Leftrightarrow (\, x \succcurlyeq y$ and $y \succcurlyeq x\,)$ for all $x, y \in S$

(**strict preference**) $x \succ y \Leftrightarrow (\, x \succcurlyeq y$ and not$(y \succcurlyeq x)\,)$ for all $x, y \in S$

(**strict aversion**) $x \prec y \Leftrightarrow y \succ x$ for every $x, y \in S$

(**incomparability**) $x \parallel y \Leftrightarrow (\,not(x \succcurlyeq y)$ and not$(y \succcurlyeq x)\,)$ for every $x, y \in S$

Preference relations defined on a finite set formally correspond with the concept of simple directed graphs (shortly digraphs). Accordingly, the graphical representation of digraphs will allow us to illustrate our investigation. For every non-empty $A \subseteq S$, the **restriction** of $\succcurlyeq$ to A is the preference relation $\succcurlyeq_{|A}$ defined as follow: $\succcurlyeq_{|A} = \{(x, y) \in A \times A$, such that: $x \succcurlyeq y\}$. By abuse, we do not specify the restriction, the context enabling to identify the targeted subset of S. A preference relation $\succcurlyeq$ is:

- **transitive** iff $[x \succcurlyeq y$ and $y \succcurlyeq z] \Rightarrow x \succcurlyeq z$, for all $x, y, z \in S$
- **quasi-transitive** iff $[x \succ y$ and $y \succ z] \Rightarrow x \succ z$, for all $x, y, z \in S$ iff the strict preference relation is transitive
- **P-acyclic** iff $\forall\, t > 2$ and $\forall\, x_1, x_2, \ldots, x_t \in S$, $[x_1 \succ x_2 \succ \ldots \succ x_t] \Rightarrow$ not$(x_t \succ x_1)$ iff $\succcurlyeq$ has no circuit of strict preference.
- an **equivalence relation** iff it is reflexive[2], symmetric and transitive
- a **partial preorder** iff it is reflexive and transitive
- a **complete** (or **total**) **preorder** iff it is reflexive, transitive and complete
- a **complete** (or **total**) **order** iff it is reflexive, transitive, antisymmetric and complete

Given a finite non-empty set S structured by a preference relation $\succcurlyeq$, the **maximal set** (or **efficient set**) of S according to $\succcurlyeq$, denoted $M(S, \succcurlyeq)$, is the subset of S verifying: $M(S, \succcurlyeq) = \{x \in S \mid \forall\, y \in S$, not$(y \succ x)\}$; while the **optimal set** of S according to $\succcurlyeq$, denoted $B(S, \succcurlyeq)$, is the subset of S verifying: $B(S, \succcurlyeq) = \{x \in S \mid \forall\, y \in S, x \succcurlyeq y\}$. Of course, there exists other choices of axioms identifying preferred (i.e. best quality, or best compromise) solutions from a preference relation, and we refer to [11, 24] for a deepening.

Given a preference relation $\succcurlyeq$ on a finite set S, another preference relation $\succcurlyeq'$ on S is an **extension** of $\succcurlyeq$ if $\forall\, x, y \in S, x \succ y \Rightarrow x \succ' y$. The relation $\succcurlyeq'$ is called a **linear extension** of $\succcurlyeq$ if $\succcurlyeq'$ is an extension of $\succcurlyeq$ and $\succcurlyeq'$ is a total order. We have the following result (see [23]): a preference relation $\succcurlyeq$ on a finite set S is P-acyclic $\Leftrightarrow$ every non empty subset of S has a non empty maximal set ($\Leftrightarrow \forall\, \varnothing \neq A \subseteq S, M(A, \succcurlyeq) \neq \varnothing$) $\Leftrightarrow$ there exists linear extensions of $\succcurlyeq$ and they are obtained by topological sorting.

2.2 Compact representations of preferences in combinatorial problems

In combinatorial practical applications, solutions are implicit: described by a set S of elementary components of a set E ($\Leftrightarrow S \subseteq \mathcal{P}(E)$). Then, it is necessary to imagine a **compact representation of preferences** for their elicitation (acquisition) and their processing; because these operations with an explicit representation – the listing of the couples $x, y \in S$ such that $x \succcurlyeq y$ – being usually intractable.

Thus, in classical combinatorial optimization, the preferences are represented by a utility function u from $\mathcal{P}(E)$ to $\mathbb{R}$ to maximize: $x \succcurlyeq y \Leftrightarrow u(x) \geq u(y)$. In multicriteria optimization based on the Pareto dominance, preferences are represented by a vector of utility functions $(u_1, \ldots, u_p)$, aggregated by the Pareto dominance: $x \succcurlyeq y \Leftrightarrow [\forall\, i \in \{1, \ldots, p\},\ u_i(x) \geq u_i(y)]$. This hierarchical aggregation will be noted $p\Sigma u\text{>}_{\text{PARETO}}$. And more generally, every aggregation of a family of p utility functions by a rule AR will be noted $p\Sigma u\text{>}_{AR}$. In artificial intelligence, numerous compact representations of preferences appeared: from CP-nets [4, 14] to constraints describing the preferential neighbourhood of the solutions (called preferential constraints in [13]), by going through soft constraints [3, 19] and dynamic CSP [26]. In the following, any compact representation of a preference relation $\succcurlyeq$ is denoted $I(\succcurlyeq)$. We will present in § 4.1 the compact representation used here for our maximal spanning trees sub-problem.

3 PREFERENTIAL CONSISTENCY AND MAXIMAL SPANNING TREES

3.1 Maximal spanning trees problems

Consider the problem of finding a *satisficing* (in the meaning of Newell & Simon [15]) maximal spanning tree. Denoted by DS(ST/CBPR/MAX), this semi-structured problem is formulated in the following way:

[2] Mention that a binary relation $\succcurlyeq$ is **symmetric** iff $x \succcurlyeq y \Rightarrow y \succcurlyeq x$, for all $x, y \in S$; **antisymmetric** iff $x \succcurlyeq y \Rightarrow$ not$(y \succcurlyeq x)$, for all $x, y \in S$ with $x \neq y$; and **complete** (or **total**) iff $x \succcurlyeq y$ or $y \succcurlyeq x$, for all $x, y \in S$ and $x \neq y$.

DS(ST/CBPR/MAX): Given an undirected graph $G = (V, E)$ and a compact representation $I(\succcurlyeq)$ of a preference relation $\succcurlyeq$ on $\mathcal{P}(E)$, the result is a feasible spanning tree which is:
(i) maximal for $(S_{ST}(G), \succcurlyeq)$, if such a solution exists, and
(ii) suited with the system of values of the user.
Otherwise, the result is 'no'.

Remark 1. DS and CBPR mean respectively decision support and crisp binary preference relation. The condition (ii) means the user via an interactive process will treat the lack of equivalence and the incompleteness between maximal solutions. This definition of problem involves that the *satisficing* solution, must be also maximal in $(S_{ST}(G), \succcurlyeq)$. In other words, the only degree of freedom let to the DSS user is the choice of a suited solution among the maximal ones. This definition refers for example to contexts where preferences have been given by the different actors of the decision problem, next aggregated in global – possibly incomplete and intransitive – preferences $\succcurlyeq$ on the solutions $\mathcal{P}(E)$ via a compact representation $I(\succcurlyeq)$; Now, an individual: the user, being able to bring efficiently forgotten preferential information at different times of the decision process, is in charge of finding the suited solution mirroring at best global preferences.

At this semi-structured problem is associated the computable problem of finding a maximal spanning tree, denoted **ST/CBPR/MAX**, the definition of which corresponds with the DS(ST/CBPR/MAX) one, after erasing the property (ii). In such a general framework, these computable problems are hard. To be convinced, it is sufficient to consider the peculiar case where the used compact representation of preferences is the Pareto-based multicriteria one. Hence, the membership problem associated with this multiobjective spanning trees problem is NP-complete [6, 12].

3.2 Preferential consistency for maximal spanning trees

In Constraint Programming [19], **consistency** is a part of a more general problematic called **description**. The aim of consistency is the description of the feasible set of a *constraint system* by way of values or combinations of values belonging to at least one feasible element.

Consistency problematic can be extended, in the framework of combinatorial problems exploiting non-conventional preferences, so as to take into account of preferential information. Simply, consistency will not rely on feasibility but on best quality or best compromise. Hence, we won't remove inconsistent values in the meaning that they belong to no feasible solution, but rather because they belong to no preferred solution. In this case, we speak about **preferential consistency**.
Without going into details, problems consisting in erasing preferentially inconsistent values, from a constraint system and a compact representation of a preference relation, are called **preferential consistency problems**. As in constraint satisfaction, several levels of preferential consistency can be defined, according to whether all or a part of preferentially inconsistent information is deleted. We named **global preferential consistency** the removing of all the preferentially inconsistent information.

Remark 2. In a non-conventional preference context, each used choice axiom (e.g. optimality, maximality, domination, …) identifies a specific choice set (optimal set, maximal set, domination set, …) which are generally pairwise different (see § 2.1). This other parameter specializes preferential consistency. Thus, we speak about **OPT-consistency** for preferential consistency using optimality as choice axiom, **MAX-consistency** for preferential consistency using maximality, and so on.

To better understand preferential consistency, in the following, we study in details the case of maximal spanning tree problem. Consider then the following general computable problem, of preferential consistency for maximal spanning trees of a graph:

GPC(ST/CBPR/MAX): Given an undirected graph $G = (V, E)$ and a compact representation $I(\succcurlyeq)$ of a preference relation $\succcurlyeq$ on $\mathcal{P}(E)$, list the edges in E belonging to a maximal spanning tree for $\succcurlyeq$, if such edges exist. Otherwise return 'no'.

An edge e is called **MAX-consistent** for $(G, I(\succcurlyeq))$ if there exists at least one maximal spanning tree for $(S_{ST}(G), \succcurlyeq)$ containing e. Otherwise, it is called **MAX-inconsistent** for $(G, I(\succcurlyeq))$.

In this article, we do not dwell on the computational complexity of this problem. But there are great chances it is at least as difficult as ST/CBPR/MAX, with the sight of investigations in constraint programming [2, 19]. Yet, in order to better appreciate the using of this kind of computable problem in a DSS, we turn towards an efficiently solvable sub-problem of ST/CBPR/MAX.

4 THE ST/TOSORT-VSMAX/MAX PROBLEM

4.1 Compact representation and TOSORT-VSMAX condition

From now, to point out an edges set, for example $\{a, b\}$, we adopt the notation ab. Given an undirected graph $G = (V, E)$ and a P-acyclic preference relation $\succcurlyeq_E$ on E, we consider the binary relation $\succcurlyeq_K$ on $\mathcal{P}(E)$ defined as follow:
$\forall\, x, y \in \mathcal{P}(E),\ x \succcurlyeq_K y \ \Leftrightarrow$
$\exists$ a linear extension $\{e_1, \ldots, e_{|E|}\}$ of $\succcurlyeq_E$ on E, verifying:
$e_i \succ_E e_j \Rightarrow i < j$ for all $1 \le i, j \le |E|$, and
for every $1 \le j \le |E|$, $e_j \notin x \ \Rightarrow\ (x \cap \{e_1, \ldots, e_{j-1}\}) \cup \{e_j\}$ contains a cycle

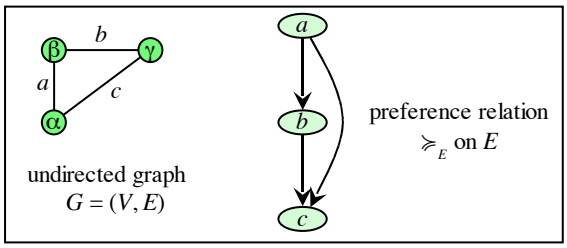


Figure 1. Example of an undirected graph and a totally ordered preference relation on its edge set[3].

Example 1. The Figure 1 illustrates the case of a complete undirected graph G on 3 vertices, with a total order $\succcurlyeq_E$ on E. Then, the binary relation $\succcurlyeq_K$ verifies (Figure 2), in addition with reflexive arcs, that: $A \succ_K B$, for every $A \in \mathcal{M} = \{ab, abc\}$ and $B \in \mathcal{P}(E) \setminus \mathcal{M}$, because the only linear extension of $\succcurlyeq_E$ is itself.

[3] To avoid surcharges of the graphical representation, the reflexive arcs are not drawn.

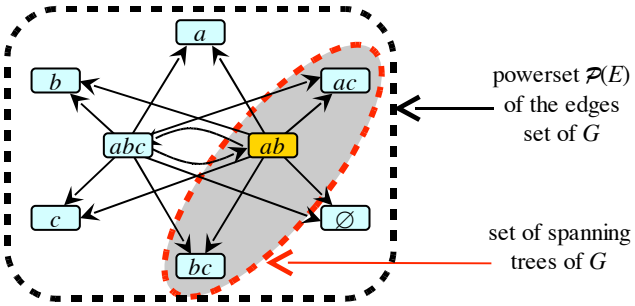


Figure 2. The relation $\succcurlyeq_K$ elaborated from $(G, \succcurlyeq_E)$ of Figure 1.

The Figure 3 considers an undirected graph $G = (V, E)$, with $V = \{\alpha, \beta, \gamma, \delta\}$ and $E = \{a, b, c, d, h\}$; and a P-acyclic relation $\succcurlyeq_E$ on the edges set E of G verifying, in addition of reflexive arcs: $a \succ_E h$, $c \succ_E b$, $c \succ_E d$, $d \succ_E b$, $h \succ_E b$, $c \simeq_E h$, $d \simeq_E h$.

Then, the binary relation $\succcurlyeq_K$ establishes a bipartition $\{\mathcal{M}, \mathcal{P}(E) \setminus \mathcal{M}\}$ of $\mathcal{P}(E)$ with $\mathcal{M} = \{x \in \mathcal{P}(E)$ such that: $acd \subseteq x$ or $ach \subseteq x\}$ and satisfies the following relations: $\forall (A, B) \in \mathcal{M} \times (\mathcal{P}(E) \setminus \mathcal{M})$, $A \succ_K B$ and $\forall (A_1, A_2) \in \mathcal{M} \times \mathcal{M}, A_1 \simeq_K A_2$.

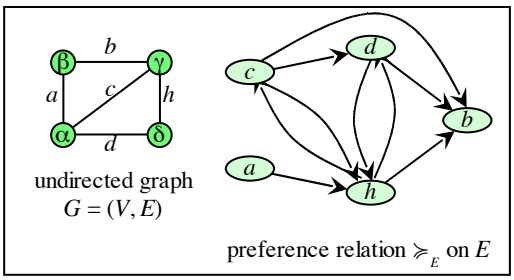


Figure 3. Example of an undirected graph and a P-acyclic preference relation on its edge set[3].

Definition 1. A preference relation $\succcurlyeq$ on $\mathcal{P}(E)$ is called **TOSORT-VSMAX** for the couple $(G, \succcurlyeq_E)$ iff: $\forall (x, y) \in S_{ST}(G) \times S_{ST}(G)$ with $x \neq y$,

$$\begin{cases} x \succ_K y \Rightarrow x \succ y & (\Leftrightarrow \text{ the relation } \succcurlyeq \text{ is an extension of } \succcurlyeq_K) \\ x \simeq_K y \Rightarrow x \simeq y \text{ or } x \parallel y \end{cases}$$

Remark 3. The word TOSORT in the notation TOSORT-VSMAX points out the relation $\succcurlyeq_K$: the relation $\succcurlyeq_E$ can be topologically sorted $\Leftrightarrow$ the relation $\succcurlyeq_E$ is P-acyclic $\Leftrightarrow$ there exists a non-empty maximal set of edges for every non-empty edges subset of E $\Leftrightarrow$ there exist total orders extending $\succcurlyeq_E$. And the second word VSMAX points out both conditions of this definition – the extension condition and the translation of the indifference of $\succcurlyeq_K$ into indifference and incomparability of $\succcurlyeq$ – which define a very strong version of maximality.

Example 2. The Figure 4 illustrates a preference relation on $\mathcal{P}(E)$ satisfying TOSORT-VSMAX for the couple $(G, \succcurlyeq_E)$ of Figure 1. This illustration shows a TOSORT-VSMAX relation may include strict preference circuits.

For the Figure 3, the feasible spanning trees set is $S_{ST}(G) = \{abd, abh, acd, ach, adh, bcd, bch, bdh\}$; Accordingly, every TOSORT-VSMAX preference relation $\succcurlyeq$ on $\mathcal{P}(E)$ satisfies:

$$\begin{cases} \forall (x,y) \in \{acd, ach\} \times S_{ST}(G) \setminus \mathcal{M}, \quad x \succ y \\ acd \text{ and } ach \text{ are either indifferent or incomparable} \end{cases}.$$

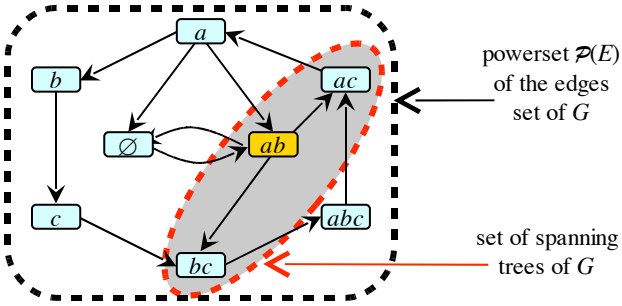


Figure 4. An example of TOSORT-VSMAX preference relation[2] on the power-set of E of Figure 1.

The preference relation $\succcurlyeq_E$ on E is called the *compact representation* of the TOSORT-VSMAX relation $\succcurlyeq$ on $\mathcal{P}(E)$. Here are some properties:

Properties 1. Given a couple $(G, \succcurlyeq_E)$ made up an undirected graph $G = (V, E)$ and a P-acyclic relation $\succcurlyeq_E$ on the edges set E, then:

(*a*) Every TOSORT-VSMAX preference relation for $(G, \succcurlyeq_E)$ identifies the same maximal set as the relation $\succcurlyeq_K$ induced by $\succcurlyeq_E$.

(*b*) The existence of feasible spanning trees warranties the existence of a non empty maximal set for $(S_{ST}(G), \succcurlyeq_K)$.

The proof is immediate. The relation $\succcurlyeq_K$ is the minimum information to know in order to identify the maximal set of TOSORT-VSMAX preference relations. Now, we consider the following sub-problem of ST/CBPR/MAX:

ST/TOSORT-VSMAX/MAX: Given an undirected graph $G = (V, E)$ and a compact representation $\succcurlyeq_E$ of a TOSORT-VSMAX preference relation $\succcurlyeq$ on $\mathcal{P}(E)$, return a maximal spanning tree for $\succcurlyeq$, if such a solution exists. Otherwise return 'no'.

We denote $S_{ST/TV/MAX}(G, \succcurlyeq_E)$ the set of possible maximal spanning trees outputted by an algorithm solving this problem.

Theorem 1. *The ST/TOSORT-VSMAX/MAX problem can be solved in a polynomial time in the input size $(G, \succcurlyeq_E)$.*

Sketch of Proof: One algorithm consists in elaborating a linear extension $\{e_1, \ldots, e_{|E|}\}$ of $\succcurlyeq_E$ on E ($\Leftrightarrow$ the TOPOLOGICAL SORT problem[4] [7, 22]); Next in assigning a utility $u(e)$ to each edge e of E in order to satisfy the following condition: $u(e_i) > u(e_{i+1})$, $1 \leq i \leq |E| - 1$; for example, $u(e_i) = |E| - i$. And, at last in solving the classic spanning tree problem ($\Leftrightarrow$ ST/Σu/OPT) with the instance (G, u). The resulting maximum spanning tree is then also a maximal solution for ST/TOSORT-VSMAX/MAX. ❑

[4] In the rest of this article, we will have to use a particular algorithm solving this problem. We will consider the following one: increasingly and greedily number the maximal edges among the not yet numbered edges of E. The designed list of edges is then a linear extension of $\succcurlyeq_E$.

4.2 Multiobjective spanning tree problems based on topological sorting

Now we confront this problem to the classical maximum spanning tree problem, and its Pareto-based multiobjective version.

Example 3. The classical problem of maximum spanning tree ($\Leftrightarrow$ ST/Σu/OPT) can be polynomially transformed into the ST/TOSORT-VSMAX/MAX problem. Indeed, for any spanning tree x of G, the sum of utilities of edges in x defines a total preorder $\succcurlyeq_u$ on $\mathcal{P}(E)$:

$$\forall (x, y) \in \mathcal{P}(E)^2, x \succcurlyeq_u y \Leftrightarrow \sum_{e \in x} u(e) \geq \sum_{e \in y} u(e)$$

The relation $\succcurlyeq_u$ is TOSORT-VSMAX, and its compact representation $\succcurlyeq^u_E$ is the preorder induced by u: $\forall e, e' \in E, e \succcurlyeq^u_E e' \Leftrightarrow u(e) \geq u(e')$. The couple $(G, \succcurlyeq^u_E)$ is then an instance of ST/TOSORT-VSMAX/MAX, and its solution set $S_{ST/TV/MAX}(G, \succcurlyeq^u_E) = B(S_{ST}(G), \succcurlyeq_u)$. This assertion is easily provable by erasing the topological sorting part of the sketch of proof of Theorem 1.

The ST/TOSORT-VSMAX/MAX problem can be used to model and solve multicriteria problems. So, the multi-attribute utility function can be aggregated first to produce global preferences on the edges, and next to partially rank sets of edges. Here is an example:

Example 4. The ST/PARETO>TOSORT-VSMAX/MAX problem considers an undirected graph $G = (V, E)$ and a couple (p, u) made up a positive number p and a multi-attribute utility function u from $E \times \{1, ..., p\}$ to $\mathbb{R}$. p is the number of considered criteria and $u(e, k)$ is the utility of the edge e according to the criterion k. In this problem, the preference information (p, u) is aggregated with Pareto dominance, in order to define a global preference relation $\succcurlyeq_{EP}$ on each edge:

$\forall e, e' \in E, e \succcurlyeq_{EP} e' \Leftrightarrow$ for every $1 \leq k \leq p, u(e, k) \geq u(e', k)$

Next, this preference relation on the edges is aggregated with the $\succcurlyeq_K$ relation, to obtain a collective opinion $\succcurlyeq_{PK}$ between the subsets of E.

Then we consider the instance $(G, (2, u))$ made up the undirected graph $G = (V, E)$ of the Figure 3, and the bicriteria utility function u given by the following table:

Table 1. Example of bicriteria utility function u(edge, criterion) on the edges of the undirected graph of the Figure 3.

	edges				
	a	*b*	*c*	*d*	*h*
criterion 1	2	2	1	1	3
criterion 2	1	1	3	2	0

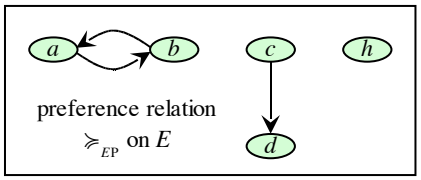


Figure 5. The preference relation $\succcurlyeq_{EP}$ on E provided by aggregation of u with the Pareto dominance[3].

By aggregating u with the Pareto dominance, we obtain the preference relation $\succcurlyeq_{EP}$ on E given by the Figure 5. At last, by solving the ST/TOSORT-VSMAX/MAX problem on this instance $(G, \succcurlyeq_{EP})$, we get the maximal set $M(S_{ST}(G), \succcurlyeq_{PK}) = \{abd, abh, acd, ach, bcd, bch\}$

Remark 4. Instead of using the Pareto dominance to obtain the global preference relation $\succcurlyeq_{EP}$ on the edges, we can apply any aggregation rule *AR* on u. The only condition on *AR* is to provide a preference relation $\succcurlyeq_{EP}$ having at least the P-acyclicity property.

In the multicriteria decision-making community [10], the multi-attribute utility function $u(e, k)$, with $(e, k) \in E \times \{1, ..., p\}$, is usually aggregated with a simple sum per criterion, to produce a family of p individual utilities on the powerset of edges. Next, this family is aggregated, generally with the Pareto dominance, into a global preference, noted in this case $\succcurlyeq_{\Sigma P}$, on the sets of edges.

Example 5. By running an algorithm solving the ST/$p\Sigma u$>PARETO/MAX problem on the instance $(G, (2, u))$ described in the Example 4, we obtain the maximal set $M(S_{ST}(G), \succcurlyeq_{\Sigma P}) = \{abh, acd, ach, bcd, bch\}$, which is strictly included in $M(S_{ST}(G), \succcurlyeq_{PK})$.

The following theorem describes the relationship between the classical hierarchical aggregation $p\Sigma u$>PARETO and ours PARETO>TOSORT-VSMAX:

Theorem 2. *Given an undirected graph G = (V, E), and a couple (p, u) made up a positive number p and a multi-attribute utility function u from E ×{1, ..., p} to ℝ; then every maximal solution for* ST/$p\Sigma u$>PARETO/MAX *is also a maximal solution for* ST/PARETO>TOSORT-VSMAX/MAX. *Formally:*

$$\forall x \in S_{ST}(G),\ x \in M(S_{ST}(G), \succcurlyeq_{\Sigma P}) \Rightarrow x \in M(S_{ST}(G), \succcurlyeq_{PK}) \quad (1)$$

Before showing this theorem, here is a lemma which describes a property of the relation $\succcurlyeq_K$:

Lemma 1. Given a couple $(G = (V, E), \succcurlyeq_E)$ and an element $x \in \mathcal{P}(E)$, then the relation $\succcurlyeq_K$ is transitive and:

$\exists y \in \mathcal{P}(E)$ such that $x \succcurlyeq_K y \Leftrightarrow x$ is optimal in $(\mathcal{P}(E), \succcurlyeq_K)$
$\Leftrightarrow x$ is maximal in $(\mathcal{P}(E), \succcurlyeq_K)$

Moreover, if $x \in S_{ST}(G)$, then:

$\exists y \in S_{ST}(G)$ such that $x \succcurlyeq_K y \Leftrightarrow x$ is optimal in $(S_{ST}(G), \succcurlyeq_K)$
$\Leftrightarrow x$ is maximal in $(S_{ST}(G), \succcurlyeq_K)$

Proof: The demonstration of the optimality (first equivalence) is immediate. What about maximality (second equivalence)? If x is optimal, then x is maximal. Now, what about the contrary case ? If x is maximal in $(\mathcal{P}(E), \succcurlyeq_K)$ then, there 2 cases:

If there exists a z such that $x \succcurlyeq_K y$, then x is optimal according to the first equivalence.

Otherwise ($\Leftrightarrow$ if such a z does not exist), then $\forall w \in \mathcal{P}(E), x \parallel_K w \Leftrightarrow \forall w \in \mathcal{P}(E)$, not$(x \succcurlyeq_K w)$ and not$(w \succcurlyeq_K x)$. Consequently, there is no optimal element in $(\mathcal{P}(E), \succcurlyeq_K)$. This assertion is equivalent to say that for every linear extension $\{e_1, \ldots, e_{|E|}\}$ of $\succcurlyeq_E$ on E, and for every subset z of E, there exists a $1 \leq j \leq |E|$ verifying that: $e_j \notin x$ and $(x \cap \{e_1, \ldots, e_{j-1}\}) \cup \{e_j\}$ is acyclic. This is possible, if and only if (V, E) is a tree and E is not in $\mathcal{P}(E)$. This is a contradiction. Hence, at last, x is optimal in $(\mathcal{P}(E), \succcurlyeq_K) \Leftrightarrow x$ is maximal in $(\mathcal{P}(E), \succcurlyeq_K)$.

The transitivity of $\succcurlyeq_K$ is a direct consequence of the first equivalence. Next, both the last equivalences are true because $(\mathcal{P}(E), \succcurlyeq_K)$ verifies the Arrow choice axiom [23]:
For any $\mathcal{A}, \mathcal{B} \in \mathcal{P}(E)$ and $\mathcal{A} \subseteq \mathcal{B}$,
If $B(\mathcal{B}, \succcurlyeq_K) \cap \mathcal{A} \neq \emptyset$ then $B(\mathcal{B}, \succcurlyeq_K) \cap \mathcal{A} = B(\mathcal{A}, \succcurlyeq_K)$
(every restriction of $\mathcal{P}(E)$ conserves the optimality). ❑

Proof (Theorem 2): First of all, both the following assertions are false:
(a) $\forall\, x, y \in S_{ST}(G),\ x \succcurlyeq_{PK} y \implies x \succcurlyeq_{\Sigma P} y$
(b) $\forall\, x, y \in S_{ST}(G),\ x \succcurlyeq_{\Sigma P} y \implies x \succcurlyeq_{PK} y$

Indeed, for the assertion (a), it is sufficient to take the undirected graph of Figure 3, with the bicriteria utility function of Table 1.

The assertion (b) is false because PK only carries out the dichotomy between the maximal set and its complementary. So, the preferences between two non maximal elements are unknown.

We prove now the formulae (1). So, we reason by contradiction: Suppose there exists an $x \in S_{ST}(G)$ maximal for $\succcurlyeq_{\Sigma P}$, but not for $\succcurlyeq_{PK}$. This proposition is equivalent with the following one, according to Lemma 1:
$\exists\, x \in S_{ST}(G)$ such that: $[\forall\, y \in S_{ST}(G),\ \text{not}(y \succ_{\Sigma P} x)]$ and
$[\forall\, y \in S_{ST}(G),\ \text{not}(x \succcurlyeq_{PK} y)]$
By definition, $\text{not}(x \succcurlyeq_{PK} y) \Leftrightarrow \exists\, e_1 \in E \setminus x$, and $\exists\, e_2 \in L(x \cup \{e_1\})$ verifying $e_1 \succ_{EP} e_2$.
Now, if we take the spanning tree y defined as follow: $y = x \cup \{e_1\} \setminus \{e_2\}$, then we have, because of the definition of $e_1 \succ_{EP} e_2$:

$$\forall\, 1 \le i \le p,\ \sum_{e \in x \setminus \{e_1\}} u(e,i) + u(e_1,i) \le \sum_{e \in x \setminus \{e_1\}} u(e,i) + u(e_2,i)\ , \text{and}$$

$$\exists\, 1 \le k \le p,\ \sum_{e \in x \setminus \{e_1\}} u(e,k) + u(e_1,k) < \sum_{e \in x \setminus \{e_1\}} u(e,k) + u(e_2,k)\ .$$

$\Leftrightarrow y \succ_{\Sigma P} x$. This contradicts the maximality of x in $(S_{ST}(G), \succcurlyeq_{\Sigma P})$. Hence the result. ❑

In the following, we propose an algorithm solving: GPC(ST/TOSORT-VSMAX/MAX), the global preferential consistency problem associated with ST/TOSORT-VSMAX/MAX.

5 GLOBAL PREFERENTIAL CONSISTENCY AND TOSORT-VSMAX

Instead of either listing all the maximal spanning trees, or finding such one tree, we will point out the removing of edges belonging to no maximal spanning tree. Especially here, we are interested in GPC(ST/TOSORT-VSMAX/MAX). Here is its definition:

GPC(ST/TOSORT-VSMAX/MAX): Let $G = (V, E)$ be an undirected graph and $\succcurlyeq_E$ be a P-acyclic preference relation on E representing a TOSORT-VSMAX preference relation $\succcurlyeq$ on $\mathcal{P}(E)$. Return all the edges of E belonging to a maximal spanning tree for $\succcurlyeq$, if such edges exist. Otherwise return 'no'.

Denote $S_{GPC(ST/TV/MAX)}(G, \succcurlyeq_E) \subseteq E$, the edges set outputted by an algorithm solving this problem. Then, by definition, we have the following equality:

$$S_{GPC(ST/TV/MAX)}(G, \succcurlyeq_E) = \bigcup_{x \in S_{ST/TV/MAX}(G, \succcurlyeq_E)} x\ . \qquad (2)$$

This equality is equivalent to the conjunction of the following assertions:
(*a*) for all $e \in S_{GPC(ST/TV/MAX)}(G, \succcurlyeq_E) \subseteq E$, there exists $x \in S_{ST/TV/MAX}(G, \succcurlyeq_E) \subseteq \mathcal{P}(E)$, such that: $e \in x$.
(*b*) for all $x \in S_{ST/TV/MAX}(G, \succcurlyeq_E) \subseteq \mathcal{P}(E)$, $x \subseteq S_{GPC(ST/TV/MAX)}(G, \succcurlyeq_E)$.

The Figure 6 presents an algorithm solving this preferential consistency problem.

```
GPCORDINALSTMAX1(G = (V, E): undirected graph, ≽_E:
                 P-acyclic preference relation on E):
                 return {edges set, no}
begin
(1)  if ( NBCONNECTEDCOMPONENTS(G) > 1 ) then return no end if
(2)  A ⊆ E ← ∅
(3)  B ⊆ E ← E
(4)  C(e) ⊆ E ← ∅, for every e ∈ E
(5)  while ( B ≠ ∅ ) do
          % loop invariants: A ∩ B = ∅  and  B ∩ C(e) = ∅
(6)     e ← CHOOSE(M(B, ≽_E))
(7)     B ← B \ {e}
(8)     C(e) ← ( ⋃_{e' ≻_E e} C(e') ∪ {e'} )
(9)     if ( NBCONNECTEDCOMPONENTS(V, C(e) ∪ {e}) <
           NBCONNECTEDCOMPONENTS(V, C(e)) ) then
           A ← A ∪ {e}
        end if
(10) end while
(11) return A
end GPCORDINALSTMAX1
```

Figure 6. An algorithm solving the GPC(ST/TOSORT-VSMAX/MAX) problem.

This algorithm supposes we know:
- Another algorithm NBCONNECTEDCOMPONENTS solving the counting problem of the connected components in an undirected graph. This problem is known solvable in a linear time (by a depth first search algorithm) for any given undirected graph (see e.g. [22, § 6.3 p. 90]).
- A choice strategy CHOOSE outputting an element of the input explicit set in the non-empty case. Otherwise, return 'no'.

Example 6. By running GPCORDINALSTMAX1 on the instance given in Figure 1 and Figure 3, we obtain as result the respective edges sets $\{a, b\}$ and $E \setminus \{b\}$.

We denote $|(G, \succcurlyeq_E)|$ the size of the instance $(G, \succcurlyeq_E)$ of ST/TOSORT-VSMAX/MAX. This size can be formulated in terms of the vertices set cardinality $m = |V|$ of graph G, the number of edges $n = |E|$ in G, and the number of arcs $p = |\succcurlyeq_E|$ in $(E, \succcurlyeq_E)$: Hence, $|(G, \succcurlyeq_E)|$ is in $O(m + n + p)$. Now, we remark that $0 \le n \le m^2$ and $0 \le p \le n^2$. Hence, $|(G, \succcurlyeq_E)|$ is in $O(m^4)$. We have the following results:

Property 2. The algorithm GPCORDINALSTMAX1 has a worst case time complexity, which is linear in the size of the input $(G, \succcurlyeq_E)$.

Proof: It is simply sufficient to see that an order of magnitude for the worst case time complexity of this algorithm GPCORDINALSTMAX1 only depends on the second loop (lines 5 to 10). The algorithm NBCONNECTEDCOMPONENTS, solving the counting connected components problem in time linear in the size of its instance (a partial graph of

G), is then in O($m + n$). Consequently, the worst case time complexity of the conditional instruction '**if …end if**' (line 9) is about $m + n$. It is similar for:

- line 6, where the choice strategy necessitates a greedy search of maximal edge, solvable in the worst case in O(n)
- line 8, where the maximum number of possible unions is about cardinality of E, i.e. in O(n).

At last, the body of the 2nd loop runs in the worst case in O($m + n$) times. Now, the number of loops is equal to the number of edges; and proves that the complexity of the algorithm GPCORDINALSTMAX1 is in O($(m + n)\cdot n$) ≈ O(m^4), i.e. linear in the input size $|(G, \succcurlyeq_E)|$. ❑

Theorem 3 *The algorithm GPCORDINALSTMAX1 returns the whole MAX-consistent edges (and only them) for maximal spanning trees of the ST/TOSORT-VSMAX/MAX problem, from an instance $((V, E), \succcurlyeq_E)$, if such trees exist. Otherwise returns 'no'.*

The logic underlying this algorithm consists in putting an edge $e \in E$ in a best scenario of choice, in order to elaborate a linear extension of $\succcurlyeq_E$ (= the minimal number assigned to e among the linear extensions). Such a best scenario consists in choosing e as soon as possible, during the topological sort. For that, the topological sorting algorithm has to number every better edge e' than e for $\succcurlyeq_E$; next the edge e' is numbered iff every better edge than e' is numbered, and so on. In the best case, when e is numbered, if the number of connected components decreases when we add e to the already numbered edges, then e can be chosen to belong to a maximal spanning tree for a TOSORT-VSMAX preference relation. Indeed, this best scenario may then be completed in a maximal spanning tree, by iteratively choosing any maximal remaining edge.

Before showing this theorem, here is a lemma which will help us in the demonstration.

Lemma 2. Given an instance $(G, \succcurlyeq_E)$, with $\succcurlyeq_E$ P-acyclic, denote $C(e)$ the edges of E for which there exists a path of strict preferences towards e: $C(e) = \{f \in E$ such that: $\exists\, f^{(1)}, \ldots, f^{(p)} \in E$, with $p \geq 0$, and $f \succ_E f^{(1)} \succ_E \ldots \succ_E f^{(p)} \succ_E e\}$. Then, for every $A \subseteq E$, $C(e) \setminus A \neq \emptyset \Rightarrow M(C(e) \setminus A, \succcurlyeq_E) \subseteq M(E \setminus A, \succcurlyeq_E)$

Proof: First of all, let us clarify the set $E \setminus C(e)$:

$E \setminus C(e) = \{f \in E$ such that: There exists no path of strict preferences from f to e in $(E, \succcurlyeq_E)\}$

$= \{f \in E$ such that: There is no path of strict preferences from f to e_1 in $(E, \succcurlyeq_E)$, $\forall\, e_1 \in C(e)\}$.

Indeed, if such a path existed from f to e_1, and – by definition of $C(e)$ – from e_1 to e, then there would exist a path of strict preferences from f to e.

Show now lemma: Suppose that $C(e) \setminus A \neq \emptyset$. Then every edge $f \in M(C(e) \setminus A, \succcurlyeq_E)$ verifies: $\forall\, f_1 \in C(e) \setminus A$, not($f_1 \succ_E f$)

$\Rightarrow$ There exists no path of strict preferences from f_1 to f in $(C(e) \setminus A, \succcurlyeq_E)$.

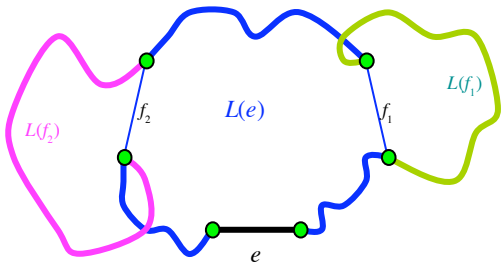
Hence, if some edges are added to $C(e) \setminus A$ – in this case $E \setminus (C(e) \cup A)$ – for which there exists no path of strict preferences from $f_2 \in E \setminus (C(e) \cup A)$ to $e_1 \in C(e) \setminus A$, then it won't also exist a path from f_2 to f. At last, $\forall\, f \in M(C(e) \setminus A, \succcurlyeq_E)$, $\forall\, f_1 \in E \setminus A$, not($f_1 \succ_E f$). This shows lemma. ❑

Proof (Theorem 3): Firstly, we have the following equivalence, because of the Properties 1 (*b*) translated by the first line of the algorithm: GPCORDINALSTMAX1$(G, \succcurlyeq_E)$ = 'no' $\Leftrightarrow S_{ST}(G) = \emptyset$.

Then point out on the first part of the proposition: Suppose GPCORDINALSTMAX1$(G, \succcurlyeq_E) \neq$ 'no', and show – by help of Lemma 2 – formulae (2):

$$\text{GPCORDINALSTMAX1}(G, \succcurlyeq_E) = \bigcup_{x \in S_{ST/TV/MAX}(G, \succcurlyeq_E)} x \quad .$$

Direct inclusion: For any $e \in$ GPCORDINALSTMAX1$(G, \succcurlyeq_E) \subseteq E$, there exists $x \in S_{ST/TV/MAX}(G, \succcurlyeq_E) \subseteq \mathcal{P}(E)$, such that: $e \in x$. Indeed, such an x can be designed by using the strategy described in the previous remark with a topological sort of $(E, \succcurlyeq_E)$. So, as long as we are not at an iteration k such that e is maximal in the set B_k of not yet numbered edges, then, during iterations $i < k$, the choice strategy consists in taking as current edge e_i a maximal edge for $(C(e) \setminus (E \setminus B_i), \succcurlyeq_E)$, with $C(e) \setminus (E \setminus B_i) \subseteq B_i$. According to the above lemma, $M(C(e) \setminus (E \setminus B_i), \succcurlyeq_E) \subseteq M(B_i, \succcurlyeq_E)$. Therefore, this strategy is available, and the iteration $k = |C(e)|$. During the iteration k, given e decreases the number of connected components in $C(e)$ – because e is in GPCORDINALSTMAX1$(G, \succcurlyeq_E)$ and then verifies the condition of line 9 in GPCORDINALSTMAX1 –, then e is chosen to be added to A_{k-1}, the current tree. Next, during iterations $i > k$, the topological sort algorithm takes as current edge, any edge of $M(B_i, \succcurlyeq_E)$. At last, the elaborated linear extension can be associated to a utility function (see sketch of proof of Theorem 1) and next used as instance of an algorithm solving ST/Σu/OPT, which necessarily returns a solution $x \neq$ 'non', containing e and then maximal for $(G, \succcurlyeq_E)$.



Legend:
Blue : $L(e)$ is an undirected path in $C(e) \subseteq E$ between both the ends of e.
f_1 and f_2 are two edges of $L(e) \setminus x$
$L(f_1)$ and $L(f_2)$ are 2 undirected paths in x between both the ends of respectively f_1 and f_2.
The edges of the bold path is included in x.

Figure 7. Illustration for demonstration of Theorem 3.

Converse inclusion: For every $x \in S_{ST/TV/MAX}(G, \succcurlyeq_E) \subseteq \mathcal{P}(E)$, $x \subseteq S_{GPC(ST/TV/MAX)}(G, \succcurlyeq_E)$. Indeed, reason by contradiction. Suppose that:

$\exists\, x \in S_{ST/TV/MAX}(G, \succcurlyeq_E)$ and $x \not\subseteq S_{GPC(ST/TV/MAX)}(G, \succcurlyeq_E)$.

$\Leftrightarrow \exists\, e \notin S_{GPC(ST/TV/MAX)}(G, \succcurlyeq_E)$ although: $e \in x$, and $x \in S_{ST/TV/MAX}(G, \succcurlyeq_E)$.

$\Leftrightarrow$ The number of connected components does not decrease if we add e in $C(e)$, according to line 9 of GPCORDINALSTMAX1

$\Leftrightarrow$ There exists in $C(e)$ an undirected path $L(e)$ between both the ends of e.

So that e should be chosen during the design of x, because $e \in x$, it is necessary that e be maximal at an iteration $k \leq |E|$, if we use the Kruskal's algorithm to solve ST/TO-SORT-VSMAX/MAX. At this iteration, $e \in A_k$, the tree at iteration k, and e decreases the number of connected components in A_{k-1}. If $e \in M(E, \succcurlyeq_E)$, then $e \in S_{GPC(ST/TV/MAX)}(G, \succcurlyeq_E)$, this is a contradiction with the initial assumption. Accordingly, $C(e) \neq \emptyset$. Moreover, every edge of $C(e)$ has already been chosen in the scenario of the topological sort algorithm during iterations $i < k$, in order that e be maximal during the iteration k. It is sure that $C(e) \not\subseteq x$ because it would exist an undirected path $L(e) \cup \{e\}$ in x; that is contradictory with x is a tree. Hence $C(e) \setminus x \neq \emptyset$. And for every $f \in C(e) \setminus x$, f has not been added to x because during the iteration $i < k$ where it has been chosen, there already exists an undirected path $L(f)$ in $A_i \subseteq x$ between the ends of f.

Now, the edge set $(C(e) \cap x) \cup \left(\bigcup_{f \in C(e) \setminus x} L(f) \right)$ is an undirected path (or contains such a path) in x between the ends of e, making up with e an undirected path in x (see Figure 7). This contradicts the assumption x is a tree. That demonstrates the converse inclusion. ❑

6 CONCLUSION AND PERSPECTIVES

One of the limits, devolved upon decision processes based on listing of preferred solutions suggested by Perny & Spanjaard [18] to solve ordinal combinatorial problems, was the intractability of large size inputs. We introduced another kind of computable problems, preferential consistency ones. Their outputs can be processed in real-time by a human being (i.e. linear in the input size). These computable problems are based both on the notion of consistency pointing out by constraint programming (CP), and on the notion of choice investigated in decision aiding (DA). In the case of maximal spanning trees problems satisfying the TOSORT-VSMAX condition, we proposed an algorithm solving the global consistency problem, with a linear worst case time complexity in its input size.

One of the aims of this article is to bring together the CP and OR-DA communities, to process more efficiently combinatorial problems exploiting complex preferences. Their mutual contributions open a new way of interactive solving of semi-structured combinatorial problems. Consequently, the perspectives are numerous:

At first, with preferential consistency: Global preferential consistency can be used in an interactive decision process, where the user makes some local decisions (choice), and where the DSS is restricted to remove preferential inconsistent domain-values. However, such support systems may not always warrant a preferred solution for the initial instance. Consequently, we have explored this way, for example by identifying domain-values which are in all preferred solutions or, by investigating rational choice theory [23] to identify some sufficient properties so that the decision process always returns a preferred solution for the initial instance, if such a solution exists.

Next, with efficient spanning trees problems and the particular compact representation of preferences used in this article: We have been scrutinizing the concept of *expressive power* of a compact representation. Any kind of compact representations models only a subset of preference relations. For example, utility functions model only total preorders. In order to better understand the type of compact representation used in this article, we focus our researches on its expressive power for spanning trees problems.

At last, with applications: What makes a good theory, it is its applicability to real world problems. The possible applications are numerous. And at this time, we work on an autonomous electrical network designing problem allowing several – not necessary cardinal – criteria. Shortly, these problems arise in isolated regions as some Pacific islands or in remote villages in rainforest. The isolation of these populations implies that the continuous supply of fossil fuels is very expensive to the community, and exorbitantly expensive if you wanted to connect to an existing electricity grid. Renewable energies form a more interesting both in terms of costs (a barrel of oil more and more expensive, and means of delivery prohibitive as boat (sometimes pirogue), helicopter or plane), in terms of noise and soil pollution, etc. These problems necessitate very complex preferential information as inhabitants opinions, cost, environmental and aesthetic criteria.

REFERENCES

[1] I.D. Aron and P. van Hentenryck, A Constraint Satisfaction Approach to the Robust Spanning Tree Problem with Interval Data. In: *18th Conference on Uncertainty in Artificial Intelligence*, Edmonton, Canada (2002)

[2] C. Bessière, E. Hebrard, B. Hnich and T. Walsh, The Complexity of Reasoning with Global Constraints, *Constraints* 12 (2), 239–259, (2007)

[3] 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, 199–240, (1999)

[4] C. Boutilier, R.I. Brafman, C. Domshlak, H.H. Hoos and D. Poole, Preference-Based Constrained Optimization with Cp-nets, *Computational Intelligence* 20, 137–157 (2004)

[5] R.E. Burkard, R.A. Cuninghame-Green and U. Zimmermann (eds.), *Algebraic and Combinatorial Methods in Operations Research*, Annals of discrete mathematics 19, North Holland, Amsterdam (1984)

[6] P.M. Camerini, G. Galbiati and F. Maffioli, The Complexity of Multi-Constrained Spanning Tree Problems, in: *Theory of Algorithms, Colloquium Pecs*, 53–101 (1984)

[7] T.H. Cormen, C.E. Leiserson and R.L. Rivest, *Introduction to Algorithms*. MIT Press and McGraw-Hill. Second Edition, Section 22.4: Topological Sort: 549–552 (2001)

[8] G. Dooms and I. Katriel, *Graph Constraints in Constraint Programming: Weighted spanning trees*, INGI research report 2006-01, UCL, Belgium (2006)

[9] G. Dooms and I. Katriel, The Minimum Spanning Tree Constraint, in: *12th International Conference on Principles and Practice of Constraint Programming*, Nantes, France (2006)

[10] M. Ehrgott and X. Gandibleux, *An Annotated Bibliography of Multiobjective Combinatorial Optimisation*, Research Report n°62/2000, Kaiserslautern University, Germany; *reprinted in OR Spektrum* 22, 425–460 (2000)

[11] J. Ghoshal, R. Laskar and D. Pillone, Topics on Domination in Directed Graphs, in: T. Haynes, S. Hedetniemi and P. J. Slater, *Domination in Graphs. Advanced topics*, coll. Monographs and textbooks in pure and applied mathematics, vol. 209, chapter 15: 401–437, Marcel Dekker, New York (1998)

[12] H.W. Hamacher and G. Ruhe, On Spanning Tree Problems with Multiple Objectives, *Annals of Operations Research* 52, 209–230 (1994)

[13] R.-R. Joseph, P. Chan, M. Hiroux and G. Weil, Decision-Support with Preference Constraints, *European Journal of Operation Research* 177 (3), 1469–1494 (2007)

[14] U. Junker, Preference-Based Problem Solving for Constraint Programming, in: G. Bosi, R. I. Brafman, J. Chomicki and W. Kießling (eds.), *Preferences 2004: Specification, inference, applications, Dagstuhl seminar proceedings*, Schloss Dagstuhl, Germany (2006)

[15] A. Newell and H.A. Simon, *Human Problem Solving*, Englewood Cliffs, NJ, Prentice-Hall (1972)

[16] T.F. Noronha, A.C. dos Santos and C.C. Ribeiro, Constraint Programming for the Diameter Constrained Minimum Spanning Tree Problem, *Electronic Notes in Discrete Mathematics* (2007)

[17] P. Perny and O. Spanjaard, Preference-Based Search in State Space Graphs, *Proceedings of the AAAI'2002 Conference*, Edmonton, Canada 751–756 (2002)

[18] P. Perny and O. Spanjaard, A Preference-Based Approach to Spanning Trees and Shortest Paths Problems, *European Journal of Operational Research* 162, 584–601 (2005)

[19] F. Rossi, P. van Beek and T. Walsh, *Hanbook of Constraint Programming*, Foundations of Artificial Intelligence series, Elsevier (2006)

[20] A. Roth and M. Sotomayor, *The Two-Sided Matching*, Cambridge University Press, Cambridge (1990)

[21] B. Roy and D. Bouyssou, *Aide Multicritère à la Décision: Méthodes et cas*, Economica, series "Gestion", Paris (1993)

[22] A. Schrijver, *Combinatorial Optimization: Polyhedra and Efficiency, Series in algorithms and combinatorics* vol. 24, Springer Verlag Publication, Chichester (2003)

[23] A.K. Sen, *Collective Choice and Social Welfare*, coll. Advanced textbooks in economics, vol. 11, Elsevier Science Publishers, Netherlands (1970)

[24] B. Subiza and J.E. Peris, Choice Functions: Rationality Re-examined, *Theory and Decision* 48, 287–304 (2000)

[25] C. Unsworth and P. Prosser, Rooted Tree and Spanning Tree Constraints, *17th ECAI Workshop on Modelling and Solving Problems with Constraints* (2006)

[26] G. Verfaillie and N. Jussien, Constraint Solving in Uncertain and Dynamic Environments: a Survey, *Constraints* 10 (3), 253–281 (2005)

[27] P. Vincke, *Multicriteria Decision Aid*, John Wiley & Sons (1992)

[28] U. Zimmerman, *Linear and Combinatorial Optimization in Ordered Algebraic Structures*, Series in Annals of Discrete Mathematics n°10, North-Holland, Amsterdam (1981)

Dynamic Symmetry Breaking Constraints

George Katsirelos[1] and **Toby Walsh**[2]

Abstract. We present a general method for dynamically posting symmetry breaking constraints during search. The basic idea is very simple. Given any set of symmetry breaking constraints, if during search a *symmetry* of one of these constraints is entailed and this is consistent with previously posted symmetry breaking constraints, then we post this constraint. The method works best with problems where symmetry can be broken with a small number of symmetry breaking constraints. We illustrate the method with two examples where a polynomial number of symmetry breaking constraints break an exponential number of symmetries. Like existing static methods for symmetry breaking, this symmetry breaking method benefits from fast and effective constraint propagation. In addition, like existing dynamic methods for symmetry breaking, this symmetry breaking methods does not conflict with the branching heuristic. Initial experimental results appear promising.

1 INTRODUCTION

Many search problems contain symmetries. For example, in scheduling problems, we can have identical orders or machines. As a second example, in workforce rostering problems, we can have equivalently skilled personnel. As a third example, in bin packing problems, we can have equal sized bins. Unless we take care, such symmetries will increase the size of the search space. In some cases, symmetries increase the size of the search space dramatically. For example, if we have n identical machines, then every schedule can be permuted into one of $n!$ symmetric schedules. If this symmetry is not factored out of search, we will waste a lot of time visiting symmetric search states.

There are a number of different methods commonly used to deal with symmetry. For example, we can *statically* add constraints before search which eliminate some or all of the symmetric solutions, or we can modify the search method so that it *dynamically* avoids symmetric solutions. Static symmetry breaking methods are simple to implement, work with any type of symmetry and tend to be highly effective. A small number of constraints can often quickly eliminate many symmetries. However, static methods have one disadvantage compared to dynamic methods: we fix in advance which solutions in each symmetry class are permitted, and branching heuristics may conflict with this choice.

In this paper, we propose a general method for posting static symmetry breaking constraints dynamically and incrementally during search. The posted symmetry breaking constraints are chosen to be consistent with the initial choices of the branching heuristic. This hybrid approach inherits good properties of both static and dynamic methods for symmetry breaking: we profit from fast propagation of the static symmetry breaking constraints, yet do not conflict with the branching heuristic. This new method is likely to be effective when either there is a small number of symmetries, or there are many symmetries but a small number of symmetry breaking constraints can break this large number of symmetries. Like other general methods for breaking symmetry, our method may be computationally expensive when there is a large number of symmetries. Our goal therefore is to identify common types of symmetry, like value interchangeability, where we require only polynomial time to break an exponential number of symmetries. Alternatively, we can apply the general method but restrict it to a polynomial number of symmetries.

[1] NICTA, Sydney, Australia, email: george.katsirelos@nicta.com.au.
[2] NICTA and UNSW, Sydney, Australia, email: tw@cse.unsw.edu.au.

2 BACKGROUND

A constraint satisfaction problem consists of a set of variables, each with a domain of values, and a set of constraints specifying allowed combinations of values for given subsets of variables. A solution is an assignment of values to variables satisfying the constraints. A common method to find a solution is backtracking search. Constraint solvers typically prune their search space by enforcing a local consistency property like domain consistency. A constraint is *domain consistent* iff for each variable, every value in its domain can be extended to an assignment that satisfies the constraint. We make a constraint domain consistent by pruning values for variables which cannot be in any solution. During the search for a solution, a constraint can become entailed. A constraint is *entailed* when any assignment of values left in the domain of the variables is a solution. For instance, $X_1 < X_9$ is entailed if and only if the largest value in the domain of X_1 is smaller than the smallest value in the domain of X_9.

Constraint satisfaction problems can contain symmetry. We will consider two special types of symmetry. A *variable symmetry* is a permutation of the variables that preserves solutions. Formally, a variable symmetry is a bijective mapping, σ of the indices of variables such that if $X_1 = d_1, \ldots, X_n = d_n$ is a solution then $X_{\sigma(1)} = d_1, \ldots, X_{\sigma(n)} = d_n$ is also. A *value symmetry*, on the other hand, is a permutation of the values that preserves solutions. Formally, a value symmetry is a bijective mapping, θ of the values such that if $X_1 = d_1, \ldots, X_n = d_n$ is a solution then $X_1 = \theta(d_1), \ldots, X_n = \theta(d_n)$ is also. More generally, symmetries can act simultaneously on variables and values. Our methods work with such general types of symmetry.

3 AN EXAMPLE

The basic idea is as follows:

> *Given any set of symmetry breaking constraints, if during search a symmetry of one of these constraints is entailed and this is consistent with previously posted symmetry breaking constraints, then we post this constraint so it holds also down all future branches.*

We illustrate this with a simple example involving just 8 symmetries. The *magic squares* problem is to label a n by n square so that the sum of every row, column and diagonal are equal (prob019 in CSPLib). A *normal* magic square contains the integers 1 to n^2. The problem has 8 symmetries corresponding to the rotations and reflections of the square. "Lo Shu", the unique normal magic square up to symmetry for $n = 3$, is an important object in ancient Chinese mathematics:

8	1	6
3	5	7
4	9	2

Consider a model with one variable for each label, an all-different constraint over all variables, and constraints that each row, column or diagonal adds up to $\frac{n^3+n}{2}$. We can rotate the solution given earlier so that the smallest corner label is at top left, and then reflect in the NW-SE diagonal so that the bottom left corner label is smaller than the top right. This eliminates all degrees of freedom.

2	7	6
9	5	1
4	3	8

We can therefore break all symmetry with the following constraints:

$$X[1,1] < X[1,n], \quad X[1,1] < X[n,1], \quad X[1,1] < X[n,n], \\ X[n,1] < X[1,n] \qquad (1)$$

Where $X[1,1]$ is the top left corner label, $X[1,n]$ is the top right, $X[n,1]$ is the bottom left and $X[n,n]$ is the bottom right.

Any of the 8 symmetries of these ordering constraints would break all symmetry. For instance, consider the variable symmetry that rotates the magic square 90° clockwise and reflects in the NW-SE diagonal. This maps $X[1,1]$ onto $X[n,1]$, $X[1,n]$ onto $X[n,n]$, $X[n,1]$ onto $X[1,1]$, and $X[n,n]$ onto $X[1,n]$. Applying this variable symmetry to (1) gives:

$$X[n,1] < X[n,n], \quad X[n,1] < X[1,1], \quad X[n,1] < X[1,n], \\ X[1,1] < X[n,n]$$

That is, the smallest corner label is now at bottom left, and the top left is smaller than the bottom right. This set of constraints would also break all symmetry.

We choose which of the 8 symmetries of (1) to use incrementally during search. For example, suppose we begin search by assigning 1 to the bottom left corner:

?	?	?
?	?	?
1	?	?

As the variables take all different values, $X[1,1] > 1$, $X[1,n] > 1$, $X[n,1] = 1$ and $X[n,n] > 1$. Hence, at this point in search, the following ordering constraints are entailed:

$$X[n,1] < X[1,n], \quad X[n,1] < X[1,1], \quad X[n,1] < X[1,n]$$

That is, the smallest corner label is at bottom left. This is a 90° rotation anti-clockwise of the first three symmetry breaking constraints given in (1). It is also a reflection in the NW-SE diagonal followed by a 90° rotation anti-clockwise of (1). At this point, we do not need to choose between these two symmetries. We simply post the three entailed ordering constraints so that they hold on all future branches. Note that the top left and bottom right corners are not yet ordered. The branching heuristic is free to choose which is smaller.

Suppose, the branching heuristics instantiates the bottom row as follows:

?	?	?
?	?	?
1	5	9

Now $X[1,1] < 9$ as variables take all-different values and $X[n,n] = 9$. Hence, at this point, the following ordering constraint is entailed:

$$X[1,1] < X[n,n]$$

That is, the top left corner is smaller than the bottom right. This ordering constraint is consistent with the three ordering constraints already posted; the four ordering constraints can be obtained by reflecting (1) in the NW-SE diagonal and then rotating 90° anti-clockwise. We therefore post this fourth ordering constraint so that it holds on all future branches. All 8 symmetries are now broken in line with the choices of the branching heuristic. Backtracking will find the unique solution with the bottom left corner smallest, and the top left smaller than the bottom right:

4	3	8
9	5	1
2	7	6

In the rest of the paper, we describe two instances of this dynamic method. In each, we post symmetry breaking constraints incrementally during search that are consistent with the choices made by the branching heuristic. Whilst we give specific examples, the method is general and can be applied to *all* types of symmetry breaking constraints. For instance, the method works with specialized symmetry breaking constraints like those used for breaking row or column symmetries [3]. It also works with general purpose symmetry breaking constraints like the "lex-leader" constraints [1].

4 INTERCHANGEABLE VALUES

We consider an example where dynamically posting symmetry breaking constraints during search is especially simple. A common type of value symmetry is when values are interchangeable. For example, when coloring the vertices in a graph, the colors (values) are interchangeable. Suppose we have n variables, X_1 to X_n taking interchangeable values from 1 to m. Based on [8], we can statically break such symmetry by converting it into variable symmetry and ordering the introduced variables. We begin by channelling into variables, Z_j representing the indices at which values first occur:

$$X_i = j \quad \Rightarrow \quad Z_j \leq i \\ Z_j = i \quad \Rightarrow \quad X_i = j$$

Where $1 \leq i \leq n$, $1 \leq j \leq m$, and $Z_j \in [1, n+m)$. We then break all symmetry by posting the ordering constraints:

$$Z_1 < Z_2 < Z_3 < \ldots < Z_m \qquad (2)$$

These ordering constraints enforce "value precedence" [5]. That is, they ensure that the first occurrence of j is before that of k for $j < k$.

Example 1 *Consider the following assignment:*

$$X_1, X_2, \ldots, X_6 \quad = \quad 1,1,2,1,3,2$$

In this case, $Z_1 = 1$, $Z_2 = 3$ and $Z_3 = 5$. Thus (2) is satisfied and the assignment obeys value precedence.

Consider, on the other hand, the symmetric assignment in which we interchange 2 *and* 3*:*

$$X_1, X_2, \ldots, X_6 \quad = \quad 1, 1, 3, 1, 2, 3$$

In this case, $Z_2 = 5$ *and* $Z_3 = 3$ *so (2) is not satisfied. This assignment therefore does not satisfy value precedence.*

As an alternative to posting (2), we can break symmetry by posting *any* symmetry of (2). To be precise, if σ is any permutation of 1 to m, we can break all symmetry by posting:

$$Z_{\sigma(1)} < Z_{\sigma(2)} < Z_{\sigma(3)} < \ldots < Z_{\sigma(m)}$$

That is, the first occurrence of $\sigma(j)$ is before that of $\sigma(k)$ for $j < k$. For instance, we could post the following symmetry of (2):

$$Z_1 < Z_m < Z_2 < Z_{m-1} < \ldots$$

This ensures 1 first occurs before m, which itself first occurs before 2, etc. It is simple to post incrementally a symmetry of (2) during search. We post the channelling constraints at the start of search as they are invariant to symmetry. Then, if at any point during search $Z_j < Z_k$ is entailed, we post $Z_j < Z_k$ so it holds on all future branches. To ensure transitivity of the Z_j variables, we maintain domain consistency on the channelling and ordering constraints. We call this $DynamicValOrder$.

Example 2 *Consider a constraint satisfaction problem with 4 interchangeable values. Suppose the branching heuristic first assigns* $X_1 = 3$*. The channelling constraints ensure* $Z_3 = 1$*,* $Z_1 > 1$*,* $Z_2 > 1$ *and* $Z_4 > 1$*. Hence* $Z_3 < Z_1$*,* $Z_3 < Z_2$ *and* $Z_3 < Z_4$ *are entailed. We therefore post these symmetry breaking ordering constraints so they hold on all future branches. These ensure that* 3 *is the first value used in any assignment.*

Suppose the branching heuristic next assigns $X_2 = 3$ *and* $X_3 = 1$*. The channelling constraints ensure* $Z_1 = 3$*,* $Z_2 > 3$ *and* $Z_4 > 3$*. Hence* $Z_1 < Z_2$ *and* $Z_1 < Z_4$ *are entailed. We therefore post these ordering constraints so they hold on all future branches. At this point:*

$$Z_3 < Z_1 < Z_2, \qquad Z_1 < Z_4$$

These constraints ensure that we only consider assignments in which 3 *is used before* 1*, and* 1 *before* 2 *and* 4*. Note that values* 2 *and* 4 *are still interchangeable. The branching heuristic is free to choose which occurs fist.*

Suppose we now backtrack. The channelling and symmetry breaking constraints leave no other choices for X_1*, and just one other choice for* X_2*, namely* $X_2 = 1$*. Other assignments are symmetric to previously considered assignments (e.g.* $X_1 = 3$*,* $X_2 = 2$ *is symmetric to* $X_1 = 3$*,* $X_2 = 1$*, whilst* $X_1 = 1$*,* $X_2 = 1$ *is symmetric to* $X_1 = 3$*,* $X_2 = 3$*).*

We prove that the $DynamicValOrder$ method breaks all symmetry. A symmetry breaking method is sound if it leaves at least one solution in each symmetry class, and complete if it leaves at most one solution.

Theorem 1 $DynamicValOrder$ *is a sound and complete symmetry breaking method for interchangeable values.*

Proof: Soundness follows quickly from the soundness of the underlying static symmetry breaking method. We have a relaxation that can only permit more assignments. Note that by maintaining domain consistency on the symmetry breaking constraints, we can always extend to a total ordering. Completeness also follows quickly from the completeness of the underlying static symmetry breaking method. Suppose we visit a complete assignment. Then we post ordering constraints for all used values. Suppose we now visit a second complete assignment that is in the same symmetry class. This contradicts one of the symmetry breaking constraints fixed by the first complete assignment. Hence, we cannot visit more than one complete assignment in each symmetry class. □

The method easily extends to partial interchangeability where values partition into equivalence classes, and values within each equivalence class are freely interchangeable. If at any point during search, $Z_j < Z_k$ is entailed where j and k are in the *same* equivalence class, then we post $Z_j < Z_k$ so it holds on all future branches.

5 VALUE PRECEDENCE

Our second example is more complex but provides additional propagation. Suppose we again have n variables, X_1 to X_n taking interchangeable values from 1 to m. As in the last section, we shall eliminate value interchangeability by enforcing a symmetry of value precedence. In [15], a global value propagator is proposed for the precedence constraint. Unlike the static method used in the last section, this propagator enforces domain consistency so prunes all possible symmetric values (see Theorem 5 in [16] for an example of symmetric values which are not pruned by the static method). The propagator in [15] uses a simple decomposition that we adapt to post symmetry breaking constraints incrementally.

We introduces $n + 1$ variables, Q_i for $i \in [0, n]$ that record the largest value used up to the index i. We set Q_i by posting:

$$Q_0 = 0, \qquad Q_i = \max(Q_{i-1}, X_i) \tag{3}$$

We then ensure value precedence by posting:

$$Q_{i+1} \quad \leq \quad 1 + Q_i \tag{4}$$

(3) and (4) break all symmetry due to interchangeable values.

Example 3 *Consider again:*

$$X_1, X_2, \ldots, X_6 \quad = \quad 1, 1, 2, 1, 3, 2$$

Then, by (3):

$$Q_0, Q_1, \ldots, Q_6 \quad = \quad 0, 1, 1, 2, 2, 3, 3$$

This satisfies (4). On the other hand, consider again the symmetric assignment in which we interchange 2 *and* 3*:*

$$X_1, X_2, \ldots, X_6 \quad = \quad 1, 1, 3, 1, 2, 3$$

Then, by (3):

$$Q_0, Q_1, \ldots, Q_6 \quad = \quad 0, 1, 1, 3, 3, 3, 3$$

This does not satisfy (4).

To post such symmetry breaking constraints incrementally during search, we take the somewhat counter-intuitive step of introducing *more* symmetry into the problem. We observe that value precedence can use any ordering on the values. For example, it could insist that the first occurrence of 3 is before that of 1, and that of 1 before that of 2. We introduce an ordering on values incrementally during search

that is consistent with the branching heuristic. To define this new ordering, we introduce m variables, P_j. The constraints will ensure $P_j = k$ if and only if the value j is in the kth position in the value precedence ordering. To break all symmetry, we post:

$$\begin{array}{llll} Q_0 = 0, & Q_i = \max(Q_{i-1}, P_{X_i}), & Q_{i+1} \leq 1 + Q_i, & \\ Q_1 = 1, & \text{ALLDIFF}(P_1, \ldots, P_m), & P_{X_i} \leq 1 + Q_{i-1} & (5) \end{array}$$

Q_i now contains the maximum *position* in the ordering defined by P_j of all the values used up to index i.

These constraints introduces $m!$ variable symmetries into the problem since the total order defined by P_j can correspond to any of the $m!$ permutations of 1 to m. For instance, one total ordering is given by:

$$P_1 = 1, P_2 = 2, P_3 = 3 \ldots, P_m = m \quad (6)$$

This will ensure 1 is the first value to occur ($P_1 = 1$), then 2 ($P_2 = 2$), then 3 ($P_3 = 3$), etc. Alternatively, we might have one of the $m!$ symmetries of (6) like:

$$P_1 = 1, P_2 = 3, P_3 = 5, \ldots, P_m = 2$$

This symmetry ensures 1 is the first value to occur ($P_1 = 1$), then m ($P_m = 2$), then 2 ($P_2 = 3$), etc.

We choose which symmetry of (6) to post incrementally during search. To do this, we maintain domain consistency on (5) and keep any prunings on the P_j when backtracking. We call this the $DynamicPrecedence$ method. The method again easily extends to partial interchangeability where values partition into equivalence classes.

Example 4 *Consider a constraint satisfaction problem with 4 interchangeable values. Suppose the branching heuristic first assigns $X_1 = 3$. From (5), we have $Q_1 = 1$ and $P_3 = 1$. As $P_3 = 1$, and P_j take all-different values, $P_1 > 1$, $P_2 > 1$ and $P_4 > 1$. Value precedence thus ensures that 3 is the first value used in any assignment. Suppose the branching heuristic next assigns $X_3 = 1$. From (5), we have $Q_2 \leq 2$, and thus $2 \leq P_1 \leq 3$. That is, the value 1 occurs 2nd or 3rd in the precedence ordering. This is to be expected. If $X_2 = 1$ or 3 then it occurs 2nd, whilst if $X_2 = 2$ or 4 then it occurs 3rd.*

Suppose we backtrack and next try $X_3 = 2$ instead. From (5), we have $2 \leq P_2 \leq 3$. That is, the value 2 also occurs 2nd or 3rd in the precedence ordering. Since we kept all prunings on P_j from the first branch, we still have $2 \leq P_1 \leq 3$. Thus P_1 and P_2 have two values between them. Propagating the all-different constraint then ensures $P_1 \in \{2, 3\}$, $P_2 \in \{2, 3\}$, $P_3 = 1$, $P_4 = 4$. At this point in search, value precedence ensures the value 3 occurs first, then 1 and 2 in either order, and the value 4 is the last of the interchangeable values to occur.

We prove that the $DynamicPrecedence$ method breaks all symmetry.

Theorem 2 *$DynamicPrecedence$ is a sound and complete symmetry breaking method for interchangeable values.*

Proof: Similar to $DynamicValOrder$. Note that by maintaining domain consistency on $\text{ALLDIFF}(P_1, \ldots, P_m)$, we can always construct a solution for the P_j. □

6 EXPERIMENTS

We implemented the symmetry breaking methods described in this paper in Gecode 2.0.1 and evaluated them on two problems: Schur numbers and graph coloring problems. Experiments were run on an 2-way Intel Xeon with 6MB of cache and 4 cores in each processor, running at 2GHz. Our hypothesis was that dynamic symmetry breaking methods would be less sensitive to the branching heuristic compared to static methods.

6.1 GRAPH COLORING

In our first experiments, we used graph coloring. Given a graph $G = \langle V, E \rangle$, we want to label each vertex $v \in V$ with a color $c(v)$, such that if $(u, v) \in E$ then $c(u) \neq c(v)$, using the smallest possible number of colors. We model this as an optimization problem with a variable for each vertex. The value of a variable is its assigned color. We post not-equals constraints among variables corresponding to neighboring vertices. All values in this problem are interchangeable. We break symmetry either with a static value precedence constraint [15] or with the $DynamicPrecedence$ method. The $DynamicValOrder$ method proved significantly slower especially on the harder problems. The results for two different value orderings, lexicographic and inverse lexicographic, are shown in the top of Table 1. All methods use the fail-first variable ordering heuristic.

We notice that the static symmetry breaking method is affected significantly by the value ordering. When using an inverse lexicographic value ordering, the static method performs uniformly worse than when using a lexicographic value ordering. The only exceptions to this are very easy instances and the instance `school1`, in which it finds a better solution. On the other hand, the dynamic method is largely unaffected by the value ordering and performs approximately the same with both branching heuristics. It is the best method in some cases, sometimes by a significant factor (e.g. `dsjc1251gb` and `school1`). In addition, it is never significantly slower that the best performing method. As predicted, the pruning from static symmetry breaking constraints can interfere with the fail first heuristic, guiding search away from easy to find solutions. In contrast, dynamic methods impose no symmetry breaking at the start of search, and thus do not prevent the branching from finding a good coloring quickly.

6.2 SCHUR NUMBERS

In our second experiment, we used problems based on Schur numbers. The Schur number $S(k)$ is the largest integer n such that $[1, n]$ can be partitioned into k sets with a, b and c placed in the same partition only if $a + b \neq c$. We turn this into a hyper-graph coloring problem by fixing n and minimizing k. We use a variable X_i for each integer $1 \leq i \leq n$, and assign $X_i = j$ iff i is placed in the j^{th} partition. Each variable's domain is therefore $[1, k]$. We post not-all-equals constraints for each triplet X_a, X_b, X_c where $a + b = c$. Clearly all values are interchangeable, as we can swap two partitions of any solution without violating any constraints. We again break symmetry either with a static value precedence constraint or with the $DynamicPrecedence$ method. Results are shown at the bottom of Table 1. As hypothesized, the performance of the dynamic method is more robust to changes in the branching heuristic than the static method. Irrespective of the branching heuristic, the dynamic method explores an almost identical search tree to the lexicographic heuristic with static symmetry breaking. By comparison, with static symmetry

Problem	Static symmetry breaking						Dynamic symmetry breaking					
	Lex			Inverse Lex			Lex			Inverse Lex		
	k	t (f/p)	b (f/p)	k	t (f/p)	b (f/p)	k	t (f/p)	b (f/p)	k	t (f/p)	b (f/p)
david	**10**	**0.09** / -	135 / -	**10**	0.44 / -	667 / -	**10**	0.42 / -	**0** / -	**10**	0.43 / -	**0** / -
dsjc1251gb	**4**	222.02 / 223.41	533031 / 536151	**4**	328.59 / 329.75	808114 / 810870	**4**	**29.97** / **31.88**	**65776** / **69766**	**4**	33.27 / 35.39	**65776** / **69766**
fullins3	**5**	**0.08** / -	96 / -	**5**	0.28 / -	520 / -	**5**	0.18 / -	**0** / -	**5**	0.17 / -	**0** / -
geom50a	**8**	1.25 / 9.18	15726 / 77246	**8**	**0.06** / 8.55	176 / 61755	**8**	0.08 / **1.32**	**0** / **6721**	**8**	0.07 / **1.32**	**0** / **6721**
miles250	**7**	**0.31** / -	242 / -	**7**	1.29 / -	1151 / -	**7**	1.41 / -	**0** / -	**7**	1.36 / -	**0** / -
myciel4	**4**	**0.01** / **0.02**	**0** / 202	**4**	**0.01** / **0.02**	38 / **162**	**4**	**0.01** / **0.02**	**0** / 188	**4**	**0.01** / **0.02**	**0** / 188
myciel5	**5**	**0.01** / 23.21	**0** / **287203**	**5**	0.05 / **23.12**	177 / 287252	**5**	0.06 / 29.22	**0** / 288622	**5**	0.06 / 29.3	**0** / 288622
r501g	**2**	**0.02** / **0.02**	**7** / **10**	**2**	0.07 / 0.07	199 / 201	**2**	0.07 / 0.07	12 / 15	**2**	0.07 / 0.07	12 / 15
r505gb	**9**	0.29 / 13.53	2196 / 100199	**9**	0.08 / 13.98	349 / 98586	**9**	**0.06** / **0.12**	**6** / **257**	**9**	0.07 / **0.12**	**6** / **257**
school1	39	5.51 / -	590 / -	27	221.33 / -	37886 / -	**21**	**56.41** / -	**0** / -	**21**	62.36 / -	**0** / -
zeroini1	**48**	**0.75** / -	**0** / -	50	8.01 / -	2921 / -	**48**	13.26 / -	**0** / -	**48**	11.46 / -	**0** / -
schur-30	**4**	2.24 / 2.38	20432 / 21091	**4**	**0.69** / **0.83**	**5024** / **5691**	**4**	2.50 / 2.64	20433 / 21095	**4**	2.51 / 2.66	20433 / 21095
schur-35	**4**	**14.58** / **14.77**	**137197** / **137859**	**4**	163.36 / 163.55	1039774 / 1040443	**4**	16.75 / 16.95	137198 / 137863	**4**	17.42 / 17.62	137198 / 137863
schur-40	6	0.05 / -	38 / -	**5**	**0.11** / -	**328** / -	6	0.05 / -	38 / -	6	0.05 / -	38 / -

Table 1. Static versus dynamic symmetry breaking. The table has four sections: static symmetry breaking constraints with lexicographic value ordering, static symmetry breaking constraints with inverse lexicographic value ordering, dynamic symmetry breaking constraints with lexicographic value ordering, and dynamic symmetry breaking constraints with inverse lexicographic value ordering. Each of the sections shows the number of colors **k** in the best solution found within the timeout, the **t**ime and the number of **b**ranches needed to **f**ind the best solution and to **p**rove optimality. "-" indicates that no solution was found (resp. optimality was not proven) within the timeout. The best results for each instance are in bold.

breaking, the inverse lexicographic heuristic is faster on `schur-30` and `schur-40`, but is less successful on `schur-35`.

7 RELATED WORK

Puget proved that symmetric solutions can be eliminated by the addition of static constraints [6]. Crawford *et al.* presented the first general method for constructing static constraints for breaking variable symmetries [1]. Their "lex-leader" method constructs a symmetry breaking constraint for each symmetry which ensures that any solution found 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. There are two weaknesses to the lex-leader method. First, it requires as many symmetry breaking constraints as symmetries. Second, it may conflict with the branching heuristic. Puget and Walsh independently extended the lex-leader method to value symmetries [11, 14]. 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 [13, 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 ordering constraints [9].

A number of dynamic methods have been proposed to deal with symmetry. For instance, SBDS posts symmetry breaking constraints dynamically during search [4]. SBDS can be seen as instance of the more general method proposed here. A limitation of SBDS is that it adds a symmetry breaking constraint for each unbroken symmetry. As there can be an exponential number of symmetries, this can be prohibitive. One of our main insights is that we can post *other* types of symmetry breaking constraint dynamically during search. A small number of symmetry breaking constraints may be adequate for special symmetries (e.g. those due to interchangeable values) or special classes of problems (e.g. problems where variables are all-different). Another dynamic method for breaking symmetry is SBDD [2]. This checks if a node of the search tree is symmetric to some previously explored node. Finally, Roney-Dougal *et al.* gave a dynamic method to construct a GE-tree, a search tree without value symmetry [12]. A weakness of both these dynamic methods is that they do not propagate their symmetry breaking constraints. It has been shown that propagation between the problem constraints and the static symmetry breaking constraints can reduce search exponentially [16].

There are a number of other approaches for posting static symmetry breaking constraints dynamically during search. Puget's STAB method adds lex-leader constraints for the stabilizers of the current partial assignment (that is, for those symmetries which are not yet broken) [7]. Since the whole symmetry group stabilizes the empty assignment, only a subset of the stabilizers can be chosen at the root when the symmetry group is large. By comparison, our method does not post any symmetry breaking constraints at the root as branching decisions have not yet forced any to be chosen. In addition, STAB only posts lex-leader constraints. One of our insights is that we can post *any* type of symmetry breaking constraint during search. Puget has also proposed dynamic lex constraints [10]. These are lex-leader like constraints in which the ordering used within the lexicographical constraint is identical to the variable ordering used by the branching heuristic. However, Puget's method performs poorly when there are large number of variable symmetries. For example, whilst a linear number of ordering constraints will break the symmetry introduced by interchangeable varialbes, Puget's method posts a symmetry breaking constraint for each of the exponential number of symmetries. In addition, unlike the method proposed here, Puget's method is limited to posting lex-leader like constraints.

8 CONCLUSIONS

We have presented a general method for dynamically and incrementally posting symmetry breaking constraints during search. The basic idea is very simple. Given any set of symmetry breaking constraints, if during search a symmetry of one of these constraints is entailed and this is consistent with previously posted symmetry breaking constraints, then we post this symmetry breaking constraint so it holds on all future branches. We illustrated the method with two examples where a polynomial number of symmetry breaking constraints can break an exponential number of symmetries. Both examples eliminate all symmetry due to interchangeable values. The first is simpler whilst the second is more complex but provides more propagation.

This hybrid approach inherits good properties of both dynamic and static symmetry breaking methods: we have fast and efficient propagation of the posted symmetry breaking constraints, yet we do not conflict with the branching heuristic. We conjecture that this new method will be effective when either there is a small number of symmetries, or there are many symmetries but only a small number of symmetry breaking constraints are needed to break symmetry. Initial experimental results appear promising. In future work, we intend to develop such hybrid methods for other special types of symmetry where a small number of constraints can break symmetry.

REFERENCES

[1] J. Crawford, G. Luks, M. Ginsberg, and A. Roy, 'Symmetry breaking predicates for search problems', in *Proceedings of the 5th International Conference on Knowledge Representation and Reasoning, (KR '96)*, pp. 148–159, (1996).

[2] T. Fahle, S. Schamberger, and M. Sellmann, 'Symmetry breaking', in *Proceedings of 7th International Conference on Principles and Practice of Constraint Programming (CP2001)*, ed., T. Walsh, pp. 93–107. Springer, (2001).

[3] P. Flener, A. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, J. Pearson, and T. Walsh, 'Breaking row and column symmetry in matrix models', in *8th International Conference on Principles and Practices of Constraint Programming (CP-2002)*. Springer, (2002).

[4] I.P. Gent and B.M. Smith, 'Symmetry breaking in constraint programming', in *Proceedings of ECAI-2000*, ed., W. Horn, pp. 599–603. IOS Press, (2000).

[5] Y.C. Law and J.H.M. Lee, 'Global constraints for integer and set value precedence', in *Proceedings of 10th International Conference on Principles and Practice of Constraint Programming (CP2004)*, pp. 362–376. Springer, (2004).

[6] J.-F. Puget, 'On the satisfiability of symmetrical constrained satisfaction problems', in *Proceedings of ISMIS'93*, eds., J. Komorowski and Z.W. Ras, LNAI 689, pp. 350–361. Springer-Verlag, (1993).

[7] J-F. Puget, 'Symmetry breaking using stabilizers', in *Proceedings of 9th International Conference on Principles and Practice of Constraint Programming (CP2003)*, ed., F. Rossi. Springer, (2003).

[8] J-F. Puget, 'Breaking all value symmetries in surjection problems', in *Proceedings of 11th International Conference on Principles and Practice of Constraint Programming (CP2005)*, ed., P. van Beek. Springer, (2005).

[9] J-F. Puget, 'Breaking symmetries in all different problems.', in *Proceedings of 19th IJCAI*, pp. 272–277. International Joint Conference on Artificial Intelligence, (2005).

[10] J-F. Puget, 'Dynamic lex constraints', in *Proceedings of 12th International Conference on Principles and Practice of Constraint Programming (CP2006)*, ed., F. Benhamou. Springer, (2006).

[11] J-F. Puget, 'An efficient way of breaking value symmetries', in *Proceedings of the 21st National Conference on AI*. Association for Advancement of Artificial Intelligence, (2006).

[12] C. Roney-Dougal, I. Gent, T. Kelsey, and S. Linton, 'Tractable symmetry breaking using restricted search trees', in *Proceedings of ECAI-2004*. IOS Press, (2004).

[13] I. Shlyakhter, 'Generating effective symmetry-breaking predicates for search problems', in *Proceedings of LICS workshop on Theory and Applications of Satisfiability Testing (SAT 2001)*, (2001).

[14] T. Walsh, 'General symmetry breaking constraints', in *12th International Conference on Principles and Practices of Constraint Programming (CP-2006)*. Springer-Verlag, (2006).

[15] T. Walsh, 'Symmetry breaking using value precedence', in *Proceedings of the 17th ECAI*. European Conference on Artificial Intelligence, IOS Press, (2006).

[16] T. Walsh, 'Breaking value symmetry', in *13th International Conference on Principles and Practices of Constraint Programming (CP-2007)*. Springer-Verlag, (2007).

Revisiting the Generalized Among Constraint

Polina Makeeva[1] and **Radoslaw Szymanek**[2]

Abstract.

This work concentrates on improving the strength and efficiency of the consistency algorithm for the general version of the Among constraint. We present an algorithm which achieves generalized arc consistency (GAC) when the general Among constraint takes the shape of some of its simpler versions. We provide a consistency algorithm with pruning strength higher than that of Among encoding using other constraints. We focus on re-using previous computation when possible and reducing the amount of work performed upon backtracking. The judicious use of trailling and re-computation contributes significantly towards the algorithm efficiency. The experimental results show that our implementation of Among achieves shorter runtimes when compared to a decomposition of Among constraint into simpler constraints.

1 Introduction

Global constraints have an essential role in Constraint Programming (CP). They allow the use of propagation algorithms based on mathematical properties of constraints. Among is a global constraint often used in the resource allocation problems, like car sequencing ([1], [8]) or rostering problems. One example is a nurse rostering problem [4] where the goal is to find a timetable for nurses in a hospital. This timetable has to satisfy constraints such as the presence of at least two nurses for every night shift and a sufficient number of days-off per week so each nurse has time to rest. In our work we are mostly interested in the extension of the Among constraint usefull for modeling the resource allocation problems when the set of resources is not known in advance. For example, the nurse rostering problem where you can hire only several nurses out of all available nurses and construct the schedule oriented on the future subset of nurses.

In our work we concentrate on a binary branching scheme in search. The motivation for the choice of binary branching are numerous. First, binary branching is commonly used in industry solvers. Second, there is a number of research work (e.g. [5]) which advocates the use of a binary branching scheme. Binary branching is more general as it does not prohibit switching to a different variable after exploring only one variable-value pair. However, it requires propagating the effects of failure. We strongly believe that the benefits of propagating the effects of failure outweighs the cost.

The remainder of this paper is organized as follows. Section 2 presents some definitions as well as a short introduction to (multi)set variables. Section 3 presents formal definitions of different versions of the Among constraint. Section 4 presents the consistency algorithm. The implementations details of this algorithm are presented in Section 5. The experimental results are presented in section 6. Finally, section 7 concludes the paper.

[1] École Polytechnique Fédérale de Lausanne (EPFL), Switzerland, email: polina.makeeva@epfl.ch

[2] Artificial Intelligence Laboratory (LIA), École Polytechnique Fédérale de Lausanne (EPFL), Switzerland, email: radoslaw.szymanek@epfl.ch

2 Formal Background

A constraint satisfaction problem (CSP) is a 3-tuple $P \triangleq \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ where $\mathcal{X}$ is a finite set of variables $\mathcal{X} \triangleq \{x_1, \ldots, x_n\}$, $\mathcal{D}$ is a set of finite domains $\mathcal{D} \triangleq \{D(x_1), \ldots, D(x_n)\}$ where the domain $D(x_i)$ is the finite set of values that variable x_i can take, and $\mathcal{C}$ is a set of constraints $\mathcal{C} \triangleq \{c_1, \ldots, c_m\}$. Each constraint c_i is defined by the ordered set $scope(c_i)$ of the variables it involves, and a set $sol(c_i)$ of allowed combinations of values. An assignment of values to the variables in $scope(c_i)$ *satisfies* c_i if it belongs to $sol(c_i)$. A *solution* to a CSP is an assignment to each variable with a value from its domain such that every constraint in $\mathcal{C}$ is satisfied.

A set variable has a domain which is a set of sets of values. We denote a set variable as S. We use multiple representation of a set variable, in particular, we use a bound representation of a set variable. The lower bound of S, denoted by lbS, contains the definitive elements. The upper bound of S, denoted by ubS, contains the potential elements of S. This representation admits S being equal to any set between lbS and ubS. For example, let the domain of the set variable S be equal to $\{\{v_1, v_2\}, \{v_2, v_3\}\}$. Then the lower bound of S is the intersection of possible values $lbS = \{v_2\}$ and the upper bound of S is a union of possible values $ubS = \{v_1, v_2, v_3\}$. Note that such a representation is weaker than the complete representation since it suggests that S has a domain equal to $\{v_2\}, \{v_1, v_2\}, \{v_2, v_3\}$, or $\{v_1, v_2, v_3\}$. We use set variables only for the purpose of the algorithm presentation. This and other representations of set variables were presented and compared in [6].

In order to discuss the strength of the propagation algorithm we introduce below the definitions used in previous research work. A constraint c_i is Generalized Arc Consistent (GAC) iff, for any variable x_i in $scope(c_i)$ assigned to any value from $D(x_i)$, there exists an assignment to all variables from $scope(c_i)$ such that this assignment belongs to $sol(c_i)$. This is the highest level of consistency a constraint can achieve. If the problem consists of only one constraint then achieving GAC implies that no wrong decision is taken during the search. Constraint c_i is Bounds Consistent (BC) iff, for any variable x_i in $scope(c_i)$, if x_i is assigned its maximum or minimum value from $D(x_i)$ then there exists an assignment to all variables in $scope(c_i)$ such that this assignment belongs to $sol(c_i)$.

3 Among Constraint

The Among constraint counts the number of X variables that take a value from a specific set of values S. The counter variable is denoted by N. We consider variable X to be *covered* by S if X takes a

value from S. The different versions of Among constraints steam from different ways of expressing S. The simplest version of Among [1] for which GAC can be obtained in polynomial time is depicted below. It represents set S as a list of integers s_i.

$$\text{Among}([X_1, .., X_n], [s_1, .., s_m], N) \text{ iff } N = |\{i|\exists j.X_i = s_j\}|$$

The algorithm presented in [3] maintains GAC on $\text{Among}([X_1, .., X_n], [s_1, .., s_m], N)$ and runs in $O(n \cdot d)$ where d is the maximum domain size. In this paper we present this algorithm only in the context of a special case of the general Among constraint.

Two generalizations were presented for the Among constraint [3]. In the first extension, the fixed list of s_i integers is replaced with a set variable S. In the second extension, instead of using a set variable S, the Among constraint employs a list of variables $[Y_1, .., Y_m]$ to specify the set of values S. From now on, we will use the term *Among constraint* to denote the second generalization of the Among constraint.

More formally, we have:

$$\text{Among}([X_1, .., X_n], [Y_1, .., Y_m], N) \text{ iff } N = |\{i|\exists j.X_i = Y_j\}|$$

This constraint can be represented with the help of the first extension:

$$\text{Among}([X_1, .., X_n], S, N) \text{ iff}$$

$$(N = |\{i|\exists j.X_i = Y_j\}|) \text{ and } S = \bigcup_Y \{Y\}$$

where S is a set variable.

The second generalization of the Among constraint represents S using a list of variables Y. In our previous example, the domain of the set variable S equals to $\{\{v_1, v_2\}, \{v_2, v_3\}\}$. Making $Y = [Y_1, Y_2]$, where $Y_1 \in \{v_1, v_3\}$, $Y_2 \in \{v_2\}$ encodes exactly that domain. As previously stated $lbS = \{v_2\}$ and $ubS = \{v_1, v_2, v_3\}$ allow S to be equal to one of the following sets $\{v_2\}$, $\{v_1, v_2\}$, $\{v_2, v_3\}$, and $\{v_1, v_2, v_3\}$. However, the representation of S using bounds and a list of Y's imposes that S can not be equal to value $\{v_2\}$ since $v_2 \notin D(Y_1)$. The value $\{v_1, v_2, v_3\}$ for S is also impossible due to the fact that there are only two Y's. The value lbS can be actually precomputed as the union of grounded Y's as depicted in Equation 1. Indeed, if $D(Y_i) = \{v_1\}$ then the value v_1 will appear in every set of the domain of S, thus, it will appear in the intersection of all possible values of S, which is exactly lbS. Moreover, equation 1 should be treated as an internal constraint within Among constraint. This equation expresses the relationship which must hold eventually. If lbS contains an element v_1 for which none of Y's is grounded to, then at least one of Y's would have to be eventually equal to v_1.

$$lbS = \cup_{Y_i \text{ is } grounded} Y_i \quad (1)$$

The algorithm for Among that uses a set variable was presented in [3] and it was shown that the level of consistency achieved by this propagation algorithm is incomparable to BC. We used this algorithm as a starting point of our implementation, where instead of S we use directly list $[Y_1, .., Y_m]$ and construct a lower bound on S (lbS) and an upper bound on S (ubS) out of Y's. The usage of Y's as the representation of S provides more accuracy than bounds representation. The bounds representation cripples the pruning strength of the consistency algorithm. Using both representations, where bounds representation is used only internally allows to strengthen the pruning capabilities of the Among constraint consistency function. The changes to any set representation are reflected on the other as soon as possible.

4 Consistency Algorithm

The consistency algorithm is presented in Algorithm 1. It consists of three parts provided as separate algorithms. The first part is concerned with pruning the domain of the X's. If the set variable S is fixed, then this part behaves exactly as the algorithm for the simple Among. It is presented in the subsection 4.1.

Algorithm 1 Consistency function

Input: X,Y,N
Output: X,Y,N
1: Alg.2(X, Y, N, lbS, ubS)
2: Alg.3(lbS,ubS,N,X)
3: Alg.4(lbS, ubS, X, Y, N, FutureDomY)

The second part of the consistency algorithm reasons about the domain of N. This algorithm, which is presented in subsection 4.2, has also its own special case when the domains of all X's are fixed. Finally, the third part of the consistency algorithm prunes the domains of the Y's and it is presented in subsection 4.3. The third part uses a variable $FutureDomY$ that is initialized at the first consistency execution to an empty domain.

In both special cases mentioned above the propagation algorithm for the Among constraint presented in this paper achieves GAC. We will use these two special situations to illustrate the functionality of the first and the second part of the consistency algorithm.

4.1 Pruning the domain of X's

4.1.1 Pruning the domain of X's when S is fixed

If during search the set variable S becomes fixed (it happens, for example, when all Y's are grounded), then the general Among constraint is transformed into a simple Among constraint because $lbS = ubS = S$. The algorithm that achieves GAC for the simple Among has been published in [3]. This algorithm does the following:

- It counts the number of X's that are already covered ($lb0$)
- It counts X's that can be potentially covered by S ($ub0$)
- It restricts N to values between $lb0$ and $ub0$
- If N is equal to $lb0$ then it subtracts S from the domains of X that can still be covered by S.
- If N is equal to $ub0$ then it intersects the domain of X's which can be potentially covered by S with S.

$ubS = lbS = \{1, 2, 3\}$
$X_1, .., X_4 \in \{1, 2\}$
$X_5, X_6 \in \{3, 4\}$

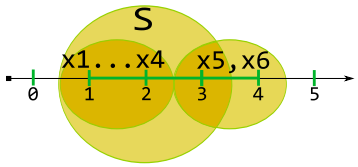


Figure 1. Set representation of the example

The example, depicted in Figures 1, 2, and 3, illustrates the behavior of the algorithm for this special case. The initial domains of X's and values for lbS and ubS are depicted by Figure 1. The circles on the right of Figure 1 are the graphical representations of the domains

of the variables. For example the set variable $S = \{1, 2, 3\}$ contains the domain of $X_1 = \{1, 2\}$, thus the circle representing variable S contains the circle that represents variable X_1. From the domains of the X's and the set variable S it can be deduced that $lb0$ is equal to 4 and $ub0$ is equal to 6.

Figure 2 presents a particular case when N is equal to $lb0$. This situation triggers pruning of the domains of the X's. Since the number of X's that need to be covered is already reached then X_5 and X_6 cannot be covered. Therefore the values of the set variable S are removed from the domains of X_5 and X_6. This pruning makes both variables X_5 and X_6 equal to 4.

Suppose $N = 4$, then we already reached the desired number of X's.
Condition ($N = lb0$) is triggered, which forces the exclusion of S from $D(X_5), D(X_6)$.
$\Rightarrow X_5 = 4, X_6 = 4$
Constraint is satisfied.

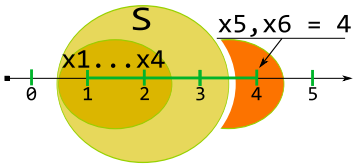


Figure 2. N is equal to $lb0$

On the other hand, Figure 3 presents a situation when N is equal to $ub0$. This situation also triggers pruning of the domains of the X's. Since all the remaining X's which are not yet covered have to be covered, then the values from the domains of X_5 and X_6 which do not belong to set variable S are removed. This makes both variables X_5 and X_6 equal to value 3.

Suppose $N = 6$, then we want to cover all possible X's. Condition ($N = ub0$) is triggered, which forces the exclusion of the complement of S from $D(X_4), D(X_5)$.
$\Rightarrow X_5 = 3, X_6 = 3$
Constraint is satisfied.

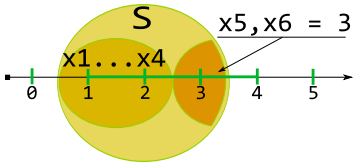


Figure 3. N is equal to $ub0$

4.1.2 *Pruning the domain of the X's. General case*

When S is not fixed it is assumed that it can take any value between the lower and upper bounds ($lbS \subseteq S \subseteq ubS$). Thus, we can use the previously described algorithm with the following modifications:

- In order to calculate the number of X's that are already covered, we count the number of X's such that $D(X) \subseteq lbS$. Indeed, the elements of lbS must be present in S, thus, X's that already belong to lbS will belong to S as well. (Alg.2 line 3)
- To calculate the number of X's that potentially might be covered we count the X's such that $D(X) \cap ubS \neq \emptyset$. This is also correct, since any element of ubS might be present in S, thus, contribute to N. (Alg.2 line 4)

Algorithm 2 Prune the domain of X

Input: X, Y, N
Output: lbS, ubS, X
1: $lbS := \cup_{Ygrounded} Y$
2: $ubS := \cup D(Y)$
3: $lb0 := |\{X_i | D(X_i) \in lbS\}|$
4: $ub0 := |\{X_i | D(X_i) \cap ubS \neq \emptyset\}|$
5: **if** $(lb0 = min(N) = max(N))$ **then**
6: **for** $X_i | D(X_i) \subseteq lbS$ **do**
7: $D(X_i) := D(X_i) \backslash lbS$
8: **end for**
9: **end if**
10: **if** $(ub0 = min(N) = max(N))$ **then**
11: **for** $X_i | D(X_i) \cap ubS \neq \emptyset$ **do**
12: $D(X_i) := D(X_i) \cap ubS$
13: **end for**
14: **end if**

The difference with the previous simpler case is that when N is equal to $lb0$, lbS is subtracted from all not yet covered X's. Moreover, if N is equal to $ub0$ then domain of each X is intersected with ubS. In all other cases for which the value of N is between $lb0$ and $ub0$ Algorithm 2 does no pruning, exactly as other algorithms previously published in the literature.

4.2 Pruning the domain of N

4.2.1 *Pruning the domain of N. X's are fixed*

If during the search all X's are fixed then only the part of the consistency algorithm which finds a proper S to cover the desired number of X's is active. The Among constraint becomes an instance of the knapsack problem where for every element $v \in ubS$ we know the exact number of X's that will be covered if we choose to include v in S. The algorithm presented in [7] achieves GAC, however it is expensive in terms of computation time. The following example will be used to present how we can transform the multiknapsack propagation algorithm to fit the special case of the Among constraint.

We construct a graph where the non horizontal edges represent the number of X's covered if the value represented by the given column is included in S. The horizontal edges in column v_i correspond to decision of not including element v_i in S. The number of X's that will be covered by v_i if v_i is included into S is denoted by $tan(v_i)$. The slope of the diagonal edge depends on $tan(v_i)$.

Figure 4 illustrates this construction. The thick lines specify the allowed edges in order to reach the weight $N \in \{5..8\}$. We see that no horizontal edge is ever taken in the column of v_1, meaning that this value must be included in S. This graph can be also used to reason about the possible domain of variable N. In this example, the value 6 must be removed from the domain of N.

This example shows that in the simple case when all X's are grounded the multiknapsack propagation algorithm is an efficient method to track the relationship between the variables Y's and N. In the next section we will analyze more general example (all X's are grounded except for one) and decide how usefull this algorithm can be in a general situation.

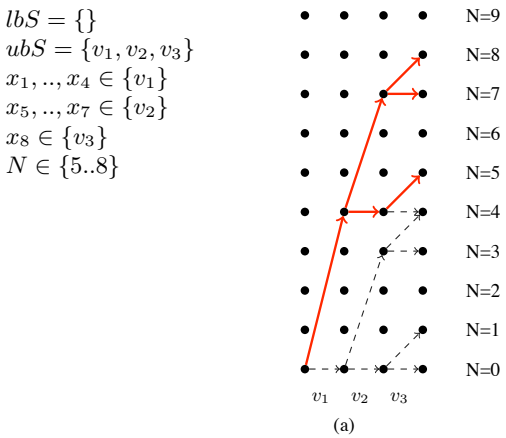


Figure 4. Knapsack graph. All X's are grounded.

4.2.2 *Prune the domain of N. General case*

For the general case, consider Figure 5. All X's are grounded except for one. The domain of x_8 has changed to $\{v_2, v_3\}$. The number of X's that will be covered if v_2 is included in S ($tan(v_2)$) is 2 or 3, and $tan(v_3)$ is 0 or 1. We clearly see that for every element in ubS, both horizontal and diagonal arcs can be taken, thus, no pruning can be done. Yet, without v_1 included in S we can cover only 4 remaining X's, thus $N \geq 5$ can not be reached.

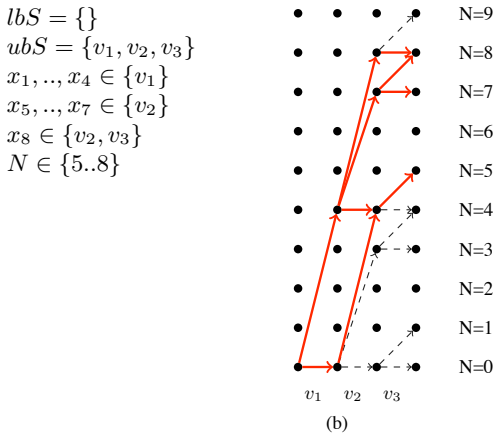


Figure 5. Knapsack graph. All X's are grounded except for one.

This example shows that the algorithm does not achieve GAC when not all X's are grounded. However, it still triggers a pruning the domain of N. Pruning the domain of N can have significant influence on the domains of other variables. It is clearly visible in case of the simple Among constraint (as it was explained in subsection 4.1), in which most propagation is triggered when N is grounded and equal to $lb0$ (Alg.2 line 5) or $ub0$ (Alg.2 line 10).

This is why we decide to change the construction of the graph in order leave the possibility to prune the domain of N. In terms of the knapsack problem, the particular situation we consider implies that the weight of any element is equal to its benefit. Therefore, it is possible to make a projection of such graphs directly on the domain of N, thus, avoiding expensive construction of the graph.

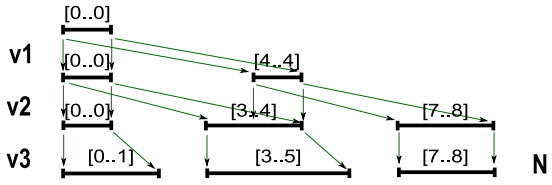


Figure 6. Projection of the knapsack graph to the N axis

Figure 6 demonstrates the projection technique applied on the previous example. First, the application of this projection technique on the elements of lbS always gives one single interval (because the decision of not including v_i into S, a horizontal arc in the knapsack graph, is not available). This interval has a lower bound equal to $lb0$, that is the number of X's that are already covered by lbS. The upper bound is the number of X's that intersect lbS, $glb0 = |\{X_i | D(X_i) \cap lbS\} \neq \emptyset|$ (Alg.3 line 2). Therefore, the potential domain of N ($potentialDomN$) is initialized to $[lb0, glb0]$(Alg.3 line 9). Afterward, the algorithm iterates through only the elements $v \in ubS \backslash lbS$ (Alg.3 line 10). For our particular example, $potentialDomN$ is initially equal to an interval $[0, 0]$ as both $lb0$ and $glb0$ are equal to zero. Unlike in the simple case when all X's are grounded, $tan(v_2)$ is now an interval $[3, 4]$. Despite $tan(v_2)$ not being determined it is possible to find lower and upper bounds for it. An upper bound is, trivially, the occurrence of v_2 in the domains of X's. $max(tan(v_2)) = occrncyVinX(v_2)$ (Alg.3 line 12) A lower bound is a number of new X's which will be definitely covered if we include v_2 into S. $min(tan(v_2)) = lb(v_2) - lb0$ (Alg.3 line 11). The main loop of Algorithm 3 computes for every element $v \in ubS \backslash lbS$ and for every interval in potentialDomN the new in-

Algorithm 3 Prune the domain of N

Input: lbS, ubS, N, X
Output: N

1: $lb0 := |\{X_i | D(X_i) \subseteq lbS\}|$
2: $glb0 := |\{X_i | D(X_i) \cap lbS\} \neq \emptyset|$
3: $ub0 := |\{X_i | D(X_i) \cap ubS \neq \emptyset\}|$
4: $min(N) := max(min(N), lb0)$
5: $max(N) := min(max(N), ub0)$
6: **if** $(max(N) < min(N))$ **then**
7: **fail**
8: **end if**
9: potentialDomN := $\{[lb0, glb0]\}$
10: **for** $(v \in ubS \setminus lbS)$ **do**
11: $lb(v) = |\{X_i | D(X_i) \in lbS \cup \{v\}\}|$
12: $occrncyVinX(v) = |\{X_i | v \in D(X_i)\}|$
13: **for** interval ∈ potentialDomN **do**
14: newMin = min(interval) $+ lb(v) - lb0$
15: newMax = max(interval) $+ occrncyVinX(v)$
16: potentialDomN = potentialDomN ∪ {[newMin, newMax]}
17: **end for**
18: **end for**
19: $D(N) := D(N) \cap potentialDomN$

terval which needs to be added to $potentialDomN$. The function $min(interval)$ and $max(interval)$ denote the minimal and maximal values within the given interval. Figure 6 depicts the execution of the main loop for our example. The potential domain of N is steadily growing. Note that if two intervals of $potentialDomN$ share the end point, then they are merged together. The worst case complexity of the algorithm for constructing $D(N)$ is equal to $O(|D(N)| \cdot |ubS|)$.

4.3 Prune the domain of Y

The construction and maintenance of the complete knapsack graph in order to perform additional pruning of the domains of the Y's variables is an expensive operation. In addition, this pruning is often weak and is weakened even more if not all the X's are grounded.

In this section we will explain pruning within lbS and ubS that uses the information calculated during the pruning of the domain of N.

For every element $v \in ubS \backslash lbS$ we ask:

- Whether the number of X's that will become covered after including v into S is greater than the upper bound of N. More formally, if $lb(v) > max(N)$ (Alg.4 line 8) then v can never enter S, thus we must remove v from every domain of Y in order to remove it from ubS.(Alg.4 line 9, 12)
- Whether there is enough X's to reach the lower bound of N if v is not covered. More formally, if $ub(v) < min(N)$ (Alg.4 line 16) then v must be present in lbS. (Alg.4 line 17)

As previously stated, lbS is computed as a union of grounded Y's (Equation1). Therefore, as soon as v_i is included in lbS, then, at least one Y_j has to be equal to v_i eventually. All such elements $v_i \in lbS$ that are not yet covered by at least one Y_j are added to a special set called *FutureDomY* (Alg.4 line18).

For each element $v_i \in FutureDomY$, the algorithm counts the occurency of v_i among domains of Y's.(Alg.4 line 23) If the occurrence of v_i is equal to zero then the constraint is in inconsistent state (Alg.4 line 28). If the occurrence of v_i is exactly one then the algorithm grounds the Y_j whose domain contains v_i to v_i, because no other Y_j can possibly take this value(Alg.4) line 24. This is followed by the removal of v_i from $FutureDomY$.

4.4 Level of consistency

The example presented in the section 4.2.2 shows that BC is not stronger than the propagation algorithm.

$$x_1, .., x_4 \in \{v1\} \qquad Y_1 \in \{v1, v3\}$$
$$x_5, .., x_7 \in \{v2\} \qquad Y_2 \in \{v2, v3\}$$
$$x_8 \in \{v2, v3\} \qquad N \in \{5..8\}$$

The algorithm prunes 6 out of the domain of N, whereas a BC algorithm will do nothing. We also saw in subsection 4.1 that in the case when all Y's are grounded general Among becomes simple Among and the propagation algorithm achieves GAC. On the other hand, the next example shows that the algorithm does not enforce BC.

$$X_1 \in \{3\},\ \ X_2 \in \{1..2\},\ \ X_3, X_4 \in \{1\}$$

$$Y_1 \in \{1..2\},\ \ Y_2 \in \{2..3\},\ \ N = 2$$

This problem has only one solution :

$$X_1 = 3,\ \ X_2 = 2,\ \ X_3, X_4 = 1,\ \ Y_1 = 2,\ \ Y_2 = 3,\ \ N = 2$$

Algorithm 4 Prune the domain of Y

Input: lbS, ubS, X, Y, N, FutureDomY
Output: Y, ubS, FutureDomY

```
1:  lb0 := |{X_i|D(X_i) ∈ lbS}|
2:  ub0 := |{X_i|D(X_i) ∩ ubS ≠ ∅}|
3:  for (v ∈ ubS \ lbS) do
4:     lb(v) = |{X_i|D(X_i) ∈ lbS ∪ {v}}|
5:     ub(v) = |{X_i|D(X_i) ∩ ubS\{v} ≠ ∅}|
6:  end for
7:  for (v ∈ ubS \ lbS) do
8:     if (lb(v) > max(N) ) then
9:        ubS := ubS\{v}
10:       for Y ∈ [Y_1, .., Y_m] do
11:          if (v ∈ D(Y) ) then
12:             D(Y) := D(Y)\{v}
13:          end if
14:       end for
15:    end if
16:    if (ub(v) < min(N) ) then
17:       lbS := lbS ∪ {v}
18:       FutureDomY := FutureDomY ∪ {v}
19:    end if
20: end for
21: for Y ∈ [Y_1, .., Y_m] do
22:    for (v ∈ FutureDomY) do
23:       if (|{Y|v ∈ D(Y)}| = 1) then
24:          D(Y) := {v}
25:          FutureDomY := FutureDomY\{v}
26:       end if
27:       if (|{Y|v ∈ D(Y)}| = 0) ) then
28:          fail
29:       end if
30:    end for
31: end for
```

The consistency algorithm will not do any pruning whereas a BC algorithm will prune 1 from X_2, as well as 1 from Y_1 and 2 from Y_2. By watching every case when the pruning can be potentially triggered we can explain why Algorithm 1 does not enforce BC.

Alg.2 :
$lbS = \{\}$
$ubS = \{1, 2, 3\}$
$lb0 = 0,\ ub0 = 4$
The pruning is not triggered because N is neither equal to $lb0$, nor to $ub0$

Alg.3 :
$PotentialDomN$ = {0..4}
No further pruning since $D(N) \subseteq PotentialDomN$.

Alg.4 :
$lb(1) = 2,\ ub(1) = 2$
$lb(2) = 0,\ ub(2) = 4$
$lb(3) = 1,\ ub(3) = 3$
The pruning is not triggered because neither condition $lb[v] > max(N)$, nor condition $ub[v] < min(N)$ is satisfied.

The difficulty of pruning 1 from the domain of X_2 lies in reasoning about the number of other X's already grounded to 1 and the value of N, which becomes too much case-specific. Algorithm 4, on the other hand, cannot prune 1 from the domain of Y_1 because it believes

in the solution in which X_3, X_4 are covered and cannot prune 2 from the domain of Y_2 because the value 3 is not yet included in $FutureDomY$.

Therefore, the level of consistency achieved by the presented propagation algorithm is incomparable to BC. At the same time the algorithm is stronger than the similar algorithm for the multiset variable S, presented in [3], which does not prune the middle values of domain of S.

Even though the question whether BC for general Among is tractable or not remains open, the series of experiments showed that on average the algorithm provides a stronger consistency function as it reduces the number of search nodes, as well as runtime when compared to an Among decomposition.

5 Iterative properties of the algorithm

This section presents how computation of different data structures can be performed iteratively. We discuss the properties of these data structures which makes reusing possible. Afterwards, we illustrate the potential of reuse on one particular example. The consistency function of Among constraint is called each time there is a change in the domain of a variable that is in the scope of the constraint. In the worst case, the consistency function is called multiple times in every node of the search tree. This makes the reuse of previously computed information crucial in order to speed up the consecutive execution of the consistency functions.

The consistency function uses extensively variables such as lbS, $lb0$, ubS, and $ub0$. In this section, we use subscripts to indicate the corresponding depth of the search level. For example, lbS_0 indicates the value of lbS at the root level (0) of the search tree. It is possible to reuse previous values for the above variables, computed at level L, when computing their values at the level L+1. The following list provides the properties of the data structures and how they can be used to speed up computation.

- $\forall L\ lbS_L \subseteq lbS_{L+1}$
  lbS can only expand with the depth of the search tree. The progressing search can only decrease the domain of Y's, therefore lbS can only grow as it collects more and more grounded Y's.
- $\forall L\ lb0_{L+1} \geq lb0_L$
  $lb0_L = |D(X)_L \subseteq lbS_L|$ increases with the depth of the search tree. Since the following holds $D(X)_{L+1} \subseteq D(X)_L$ and $lbS_L \subseteq lbS_{L+1}$ then $lb0_{L+1} \geq lb0_L$
- $\forall L\ ubS_{L+1} \subseteq ubS_L$
  ubS is equal to the union of Y's. The progressing search can only decrease the domain of Y's therefore ubS can only decrease with the depth of the tree.
- $\forall L\ ub0_L \geq ub0_{L+1}$
  The progressing search can only decrease the number of X's that intersect with ubS, as ubS and domain of X's can only shrink. Therefore, $ub0$ can only decrease with the depth of the tree.

Based on these properties of the $lbS, lb0, ubS, ub0$ we make a following conclusion. If some X_i in some search node at depth L had its domain $D(X_i)_L \subseteq lbS_L$ then for any child node this relation will hold. Thus, we do not have to re-check it in any child node. Similarly, if the relation $D(X_j)_L \cap ubS_L = \emptyset$ holds at level L it will hold for any child nodes.

We can, thus, keep track of such variables X_i and recalculate $lb0$ and $ub0$ only for the remaining variables, which we call active X's. In order to remember which X's do need to be re-checked, we could either put them into some special list (but that would use too much memory), or make the separation in the original list of $[X_1, X_2...., X_n]$ by moving variable X_i, which belongs to lbS, to the left side of the array. On the other hand, variables X_j, that do not intersect ubS, can be moved to the right side of the array. The active X's, which can influence $lb0$ or $ub0$, will be grouped in the middle of the array starting from the position $lb0 + 1$ to position $ub0$. Now, in order to calculate $lb0_{L+1}$ we calculate $lb0$ only for the active X's and add it to $lb0_L$. To calculate $ub0_{L+1}$ we calculate the number of X's that do not intersect ubS among active X's and subtract it from $ub0_L$.

Note that this requires storing values of $lb0$ and $ub0$ for every level of the search tree, which has only a constant cost. The worst case complexity of computing the mentioned data structures in a search node remains O(n). However, the constant in front is reduced. Imagine a child node $L + 1$ on the left branch of the decision tree. It has $lb0_{L+1} = lb0_L + 1$, and $ub0_{L+1} = ub0_L$ with an X_i such that $D(X_i)_{L+1} \subseteq lbS_{L+1}$ placed on the position $lb0_L + 1$. If the process backtracks to depth L then the consistency algorithm uses $lb0_L$ and $ub0_L$ to determine active X's therefore X_i becomes active again, since it is placed in between $lb0_L$ and $ub0_L$.Therefore, the backtracking requires only the restoration of old values for $lb0$ and $ub0$.

The following figures will illustrate the reuse of previous computations on the simple example used earlier. Figure 7 presents the initial values for the input parameters of the Among constraint as well as computed values for internal data structures.

$x_1, .., x_4 \in \{v1\}$ $\quad$ $Y_1 \in \{v1, v3\}$
$x_5, .., x_7 \in \{v2\}$ $\quad$ $Y_2 \in \{v2, v3\}$
$x_8 \in \{v2, v3\}$ $\quad$ $N \in \{5..8\}$

Figure 7. Initial values

At the first search node, Algorithm 3 prunes the value 6 out of the domain of N. Algorithm 4 concludes that without value v1 included in lbS it is not possible to reach the minimal bound of N, so v1 is included in lbS as well as Y_1 is assigned value v1. Figure 8 presents the value of different parameters as well as the order of X's after executing the consistency function at the first search node.

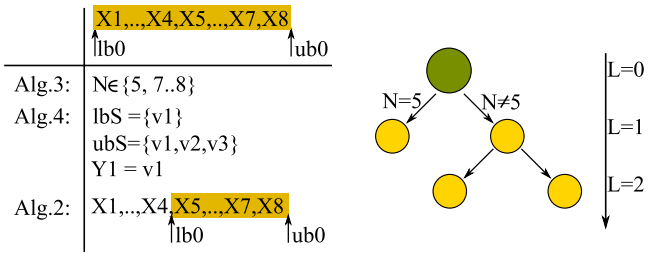


Figure 8. Search node 1

Figure 9 presents the inferences made by the consistency function while in the second search node. In this node, the algorithm initially works with four active X's : $X_5, .., X_8$. The search has grounded

variable N to value 5. Therefore, the algorithm 4 concludes that value v2 must be excluded from ubS, thus, Y_2 becomes immediately equal to v3. The value v2 was removed from ubS because if it were included in lbS then it would cover X_5, X_6, and X_7, which (together with $X_1, .., X_4$) makes $N > 5$. The variables are rearranged and X_5, X_6, and X_7 are placed after position $ub0 = 5$. This is simply done by swapping X_5 and X_8 and changing value of $ub0$ to 5. Then Algorithm 2 enters the special case when $N = ub0$, thus, it grounds X_8 to v3. The search continues to find other solutions therefore it backtracks to the previous search level.

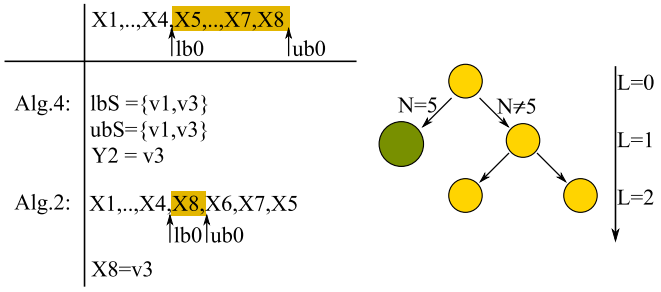


Figure 9. Search node 2

After backtracking, the current state of the search and variables is depicted in Figure 10. The right child node at the level 1 uses X's from position 5 to 8, but in the order determined by the previous consistency function executions : X_8, X_6, X_7, X_5. Trivially, the reordering does not affect the result of the algorithms. Algorithm 4 concludes that without value v2 included into lbS the lower bound of N cannot be reached. Value v2 is included in lbS and must be covered by some Y. The occurrence of value v2 among Y's is equal to one, which means that only Y_2 can be equal to v2. Therefore, Algorithm 4 grounds Y_2 to v2. The regrouping technique is applied again, variables X_6, X_7, and X_5 are swapped one by one with the only remaining active variable X_8. Each swap increases the value of $lb0$. The obtained order of X's is as follows : $X_1..X_4, X_6, X_7, X_5, X_8$.

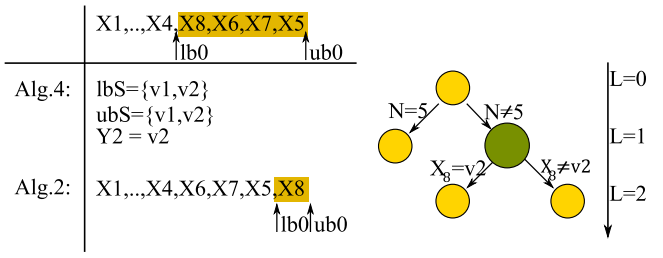


Figure 10. Search node 3

The search nodes four and five are depicted in Figure 11. In both cases, only one active X remains. The fourth search node assigns X_8 to v2 which leads to N being grounded to value 8. On the other hand, the fifth search nodes remove v2 from the domain of X_8 making N equal to 7. In both search nodes, as soon as X_8 is assigned a value it is no longer active, making $lb0$ and $ub0$ equal indicating that the constraint is satisfied.

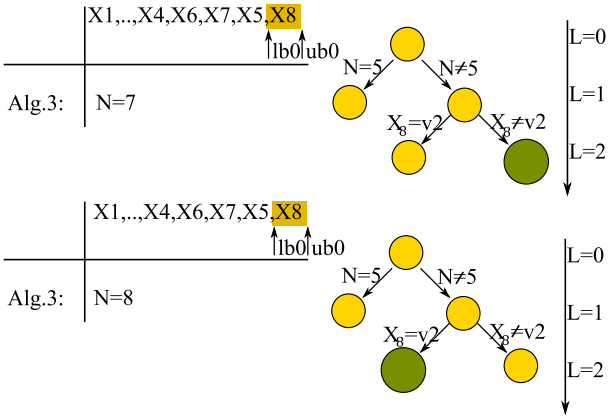


Figure 11. Search node 4 and 5

This section, up to now, has only discussed about better organization of X's variables. A similar approach can be used for variables Y's. These variables may influence only the sets lbS, ubS and $FutureDomY$. lbS is a union of grounded $Y's$, which cannot change any further, and the rest of Y's are active. It is possible to organize the active Y's to be in the right side of the array as was done for the X's.

On the other hand ubS is a union of domains of all Y's. If value v is pruned out of the domain of Y_j and there exists another Y_k such that $v \in D(Y_k)$, then ubS stays unchanged and the execution of the consistency function is not necessary. The $FutureDomY$ is a union of v such that the occurrence of v in Y's is more than 1. If value v is pruned out of the domain of Y_j then the algorithm needs to check again if there are more than one variable equals to value v. The consistency algorithm does not compute set ubS neither set $FutureDomY$ in an iterative manner. It always recomputes them even if they could be computed iteratively by maintaining the occurrence representation of the sets. We choose to recompute these sets to reduce overall time, especially visible in case of ubS.

5.1 Detaching a variable from a constraint

By default, the consistency algorithm of a constraint is called as soon as the domain of one of the constraint variables is pruned. If the change to the variable domain does not cause any further pruning then there was no benefit in executing the consistency function. There are number of techniques to discover situations when there is no further pruning possible without the need to execute the consistency function.

We consider variable X_i to be *attached* to constraint C_j if $X_i \in scope(C_j)$. Any changes to $D(X_i)$ force the re-execution of the consistency algorithm. On the other hand, if in search node s the variable X_i is *detached* from the constraint C_j, then, for the whole

search subtree rooted at s the change of $D(X_j)$ will not cause the re-execution of the consistency function. Upon backtracking to the search node above s the variable x will be reattached back to the constraint. To gain in efficiency it is important to recognize such variables as soon as possible. As it was discussed in the previous sections, if $D(X_i)$ at search node s belongs to lbS then for any child node of s this relationship will hold. Therefore, X_i can be safely detached from the constraint. Similarly, if the domain of some X_i does not intersect ubS then it can be detached from the constraint.

6 Experimental Results

In this section we present the results of experiments. We compare the computation time and the number of search nodes of Among versus an Among decomposition. Both approaches, the Among and the Among decomposition, were implemented/evaluated within JACOP framework for number of different setups. Each problem instance was solved using randomly generated order of variables. The same order was used for both solving approaches but the order of variables could differ across different instances as for each instance a random generation was performed. All experiments have been performed on the same hardware, namely a Pentium 4, and runtimes were collected as a total number of CPU-seconds that the process spent in user mode (measured with the help of the time command in RedHat Linux).

We encode Among with the following decomposition. It employs a number of constraints (e.g. Sum, Reified, Max) as well as additional variables (e.g B_i):

$$\text{Among}([X_1, ..., X_n], [Y_1, ..., Y_m], N) \text{ iff}$$

$$\forall i \in \{1, .., n\} \;\; j \in \{1, .., m\} B_{ij} = 1$$

$$\iff X_i = Y_j \in S \wedge \sum_{i \in \{1,..,n\}} (max_{j \in \{1..m\}} B_{ij}) = N$$

where each B_i is a Boolean variable indicates if X_i is equal to Y_j.

6.1 Increasing the number of X's

In the first series of experiments the problems were generated randomly with the sequence of variables $[X_1, X_2, ..., X_n]$, $[Y_1, Y_2, ..., Y_m]$ where m - (the number of Y's) remain constant (m = 5) and n - the number of X's increased from 3 to 25 (3,5,7,...,25). Each domain of X's and Y's consists of 2 random intervals and the total size of the domain never exceeds 7 elements. The domain of variable N consists of 2 or 3 random intervals drafted from the domain $\{0..n\}$. Each combination of parameters was tested 10 times with different random seeds giving 120 experiments.

Figure 12 presents the number of search nodes needed to solve a problem instance with Among constraint versus Among decomposition. A cross (+) corresponds to one problem instance. The y-axis indicates the number of the search nodes of the Among approach and the x-axis indicates the number of the search nodes for Among decomposition. Both axes are plotted in a logarithmic scale. The diagonal solid line corresponds to the function $y = x$. The crosses situated on the diagonal line correspond to the problem instances that had the same number of search nodes in both, Among constraint and Among decomposition approaches. The dashed line corresponds to the function $y = x/2$. All the crosses situated under this line have at least 2 times fewer search nodes in the search tree for Among than the Among decomposition approach. The dot-dashed line corresponds to the function $y = x/7$. All the crosses under this line indicate instances for which the Among approach was at least seven times more efficient in terms of search nodes than the Among decomposition approach.

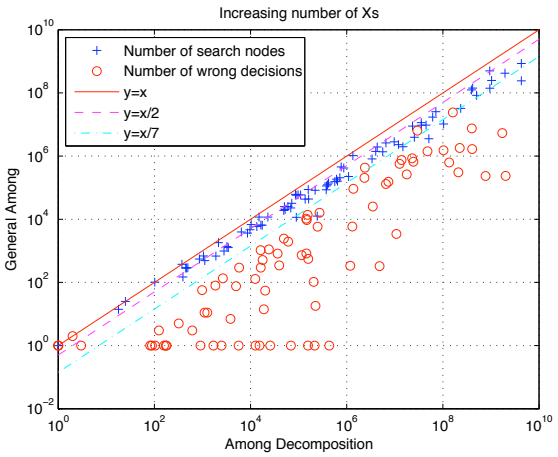


Figure 12. Number of nodes needed to find all solutions to the problem

Because of the logarithmic scale the cross distribution is close to the diagonal. Yet, only in 28.5% of the tests, Among was less than 2 times better than the decomposition. In 63% of tests, Among was at least 2 times better but less than 7 times better than the decomposition and in 8.5% of the tests Among was 7 times better than the decomposition. In order to compare the strength of propagation for Among and the Among decomposition we also plot the number of wrong decisions (round points) for the same set of problems. A wrong decision corresponds to an inconsistent leaf search node. In our experiments we used the definitions of search node, backtrack, and wrong decision as it was presented in [2]. Due to the logarithmic scale that ignores values 0 we have added value 1 to all wrong-decision numbers. Figure 12 shows that in 26% of the tests the general Among reaches GAC (value 10^0 corresponds to zero wrong decisions). Generally, the number of wrong decisions is significantly smaller for Among. There are instances that had 6 orders of magnitude more wrong decisions for the decomposition. The Among decomposition had zero wrong decision only in 3% of the tests.

Figure 13 presents the execution time of Among versus the Among decomposition. The y-axis indicates the number of seconds needed to solve the problem with the Among constraint, while the x-axis indicates the number of seconds needed to solve the problem with the Among decomposition. In 34.5% of the tests Among was slower than the Among decomposition. However, Among constraint needs more time than the Among decomposition only for small problems, which require less than 5 seconds to find all solutions. As soon as the number of variables increases the advantage of using Among versus the Among decomposition is clearly visible. In 33.4% of the tests Among was faster than the decomposition but less than 2 times faster. In 24.6% of the tests Among was at least 2 times faster and at most 7 times faster than the decomposition and in 7.5% of the tests general Among was 7 times faster than the decomposition.

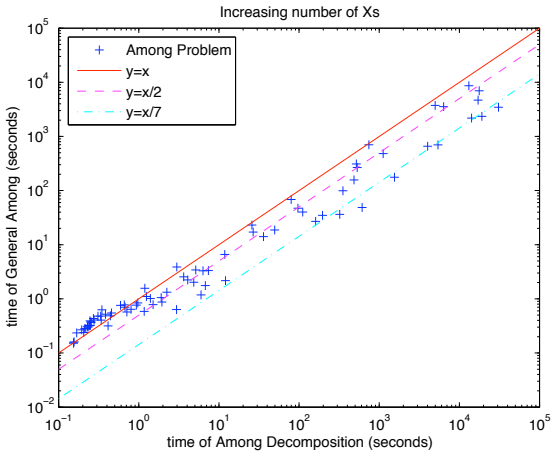


Figure 13. CPU time needed to find all solutions

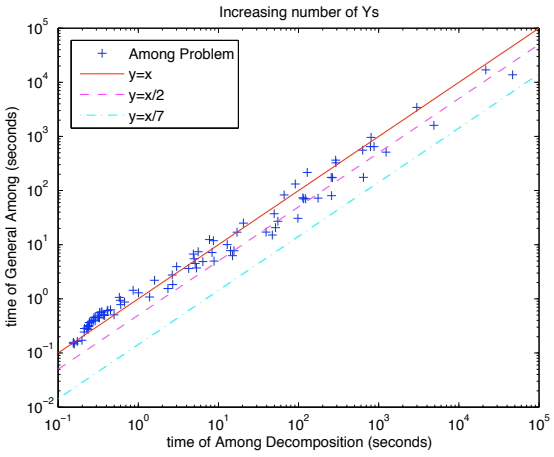


Figure 15. CPU time needed to find all solutions

6.2 Increasing the number of Y's

In the second series of experiments we kept the number of X's constant (=5) and increased the number of Y's from 3 to 25 (3,5,7,..25). The results of the experiments are presented in Figure 14. Similarly to the first series of experiments, a "+" point corresponds to one problem instance. The y-axis indicates the number of search nodes of Among and the x-axis indicates the number of search nodes of the Among decomposition.

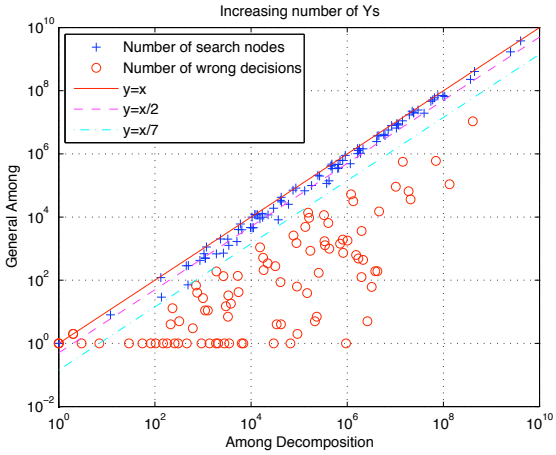


Figure 14. Number of nodes needed to find all solutions

Figure 14 shows that the problems with increasing number of Y's are harder than those from the first series of experiments. In 76.5% of the tests, Among had at most 2 times fewer search nodes than the Among decomposition. The remaining instances were solved by Among with up to 7 times fewer search nodes than the Among decomposition. The circles represent the number of wrong decisions (plus one, due to the logarithmic scale). In 30% of the tests Among reached GAC, while the Among decomposition reached GAC only in 3.5% of the tests.

Figure 15 shows the execution time of Among versus the Among decomposition. In 64% of the tests Among was worst than the decomposition. However, again the majority of instances which are solved faster by the Among decomposition are simple and can be solved relatively fast. In 36% of the tests Among was better and in around 10% of the tests Among was at least twice better than the decomposition. These results show that in case of increasing number of *Y*'s, the constraint becomes less tight as there is a less dramatic decrease in search nodes despite maintaining the significant reduction in wrong decisions.

7 Conclusions

In this work, we presented the design and implementation details of the Among constraint. Our implementation was compared to an Among decomposition for a wide range of different problem instances. We showed how to apply different techniques to improve the efficiency of the consistency function. Our studies shows that, even without reaching BC, the proposed consistency algorithm does an efficient pruning and for difficult problems it wins significantly against the Among decomposition. Moreover, it is quite common that the consistency function achieves GAC as the search tree contains no wrong decisions, while looking for all solutions.

REFERENCES

[1] N. Beldiceanu and E. Contjean, 'Introducing global constraints in CHIP', *Mathematical and Computer Modelling*, **12**, 97–123, (1994).

[2] C. Bessiere, B. Zanuttini, and C. Fernandez, 'Measuring search trees', in *Proceedings ECAI'04 Workshop on Modelling and Solving Problems with Constraints, Valencia, Spain*, ed., B. Hnich, (2004).

[3] Christian Bessiere, Emmanuel Hebrard, Brahim Hnich, Zeynep Kiziltan, and Toby Walsh, 'Among, common and disjoint constraints', **3978**, 29–43, (2006).

[4] B. M. W. Cheng, J. H. M. Lee, and J. C. K. Wu, 'A constraint-based nurse rostering system using a redundant modeling approach', in *ICTAI '96: Proceedings of the 8th International Conference on Tools with Artificial Intelligence (ICTAI '96)*, p. 140, Washington, DC, USA, (1996). IEEE Computer Society.

[5] Joey Hwang and David G. Mitchell, '2-way vs. d-way branching for csp.', in *CP*, ed., Peter van Beek, volume 3709 of *Lecture Notes in Computer Science*, pp. 343–357. Springer, (2005).

[6] Zeynep Kiziltan and Toby Walsh, 'Constraint programming with multisets', in *Proceedings of the 2nd International Workshop on Symmetry in Constraint Satisfaction Problems (SymCon-02)*, (2002).

[7] Michael A. Trick, 'A dynamic programming approach for consistency and propagation for knapsack constraint', (2001).

[8] Edward Tsang, 'Foundations of constraint satisfaction', *Journal of Information Processing Society of Japan*, **37**(1), 96–97, (1993).

Flow-Based Propagators for the SEQUENCE and Related Global Constraints[1]

Michael Maher[2] and **Nina Narodytska**[3] and **Claude-Guy Quimper**[4] and **Toby Walsh**[5]

Abstract. We propose new filtering algorithms for the SEQUENCE constraint and several extensions which are based on network flows. Our propagator for the SEQUENCE constraint enforces domain consistency in $O(n^2)$ time down a branch of the search tree. This improves upon the best existing domain consistency algorithm by a factor of $O(\log n)$. The flows used in these algorithms are derived from a linear program. Some of them differ from the flows used to propagate global constraints like GCC since the domains of the variables are encoded as costs on the edges rather than capacities. Such flows are efficient for maintaining bounds consistency over large domains and may be useful for other global arithmetic constraints.

1 Introduction

Graph based algorithms play a very important role in constraint programming, especially within propagators for global constraints. For example, Regin's propagator for the ALLDIFFERENT constraint is based on a perfect matching algorithm [14], whilst his propagator for the GCC constraint is based on a network flow algorithm [15]. Both these graph algorithms are derived from the bipartite value graph, in which nodes represent variables and values, and edges represent domains. For example, the GCC propagator finds a flow in such a graph in which each unit of flow represents the assignment of a particular value to a variable. In this paper, we identify a new way to build graph based propagators for global constraints: we convert the global constraint into a linear program and then convert this into a network flow. These encodings contain several novelties. For example, variables domain bounds can be encoded as costs along the edges. We apply this approach to the SEQUENCE family of constraints. Our results widen the class of global constraints which can be propagated using flow-based algorithms. We conjecture that these methods will be useful to propagate other global constraints.

2 Background

A constraint satisfaction problem (CSP) consists of a set of variables, each with a finite domain of values, and a set of constraints specifying allowed combinations of values for subsets of variables. We use capital letters for variables (e.g. X, Y and S), and lower case for values (e.g. d and d_i). A solution is an assignment of values to the variables satisfying the constraints. Constraint solvers typically explore partial assignments enforcing a local consistency property using either specialized or general purpose propagation algorithms. A *support* for a constraint C is a tuple that assigns a value to each variable from its domain which satisfies C. A *bounds support* is a tuple that assigns a value to each variable which is between the maximum and minimum in its domain which satisfies C. A constraint is *domain consistent* (*DC*) iff for each variable X_i, every value in the domain of X_i belongs to a support. A constraint is *bounds consistent* (*BC*) iff for each variable X_i, there is a bounds support for the maximum and minimum value in its domain. A CSP is DC/BC iff each constraint is DC/BC. A constraint is *monotone* iff there exists a total ordering $\prec$ of the domain values such that for any two values v, w if $v \prec w$ then v is substitutable for w in any support for C.

We also give some background on flows. A *flow network* is a weighted directed graph $G = (V, E)$ where each edge e has a capacity between non-negative integers $l(e)$ and $u(e)$, and an integer cost $w(e)$. A *feasible flow* in a flow network between a source (s) and a sink (t), (s,t)-flow, is a function $f : E \rightarrow \mathbb{Z}^+$ that satisfies two conditions: $f(e) \in [l(e), u(e)]$, $\forall e \in E$ and the *flow conservation* law that ensures that the amount of incoming flow should be equal to the amount of outgoing flow for all nodes except the source and the sink. The *value* of a (s,t)-flow is the amount of flow leaving the sink s. The *cost* of a flow f is $w(f) = \sum_{e \in E} w(e)f(e)$. A *minimum cost flow* is a feasible flow with the minimum cost. The Ford-Fulkerson algorithm can find a feasible flow in $O(\phi(f)|E|)$ time. If $w(e) \in \mathbb{Z}$, $\forall e \in E$, then a minimum cost feasible flow can be found using the successive shortest path algorithm in $O(\phi(f)SPP)$ time, where SPP is the complexity of finding a shortest path in the residual graph. Given a (s,t)-flow f in $G(V,E)$, the *residual graph* G_f is the directed graph (V, E_f), where $E_f = \{e \text{ with cost } w(e) \text{ and capacity } 0..(u(e) - f(e)) \mid e = (u,v) \in E, f(e) < u(e)\} \bigcup \{e \text{ with cost } -w(e) \text{ and capacity } 0..(f(e) - l(e)) \mid e = (u,v) \in E, l(e) < f(e)\}$. There are other asymptotically faster but more complex algorithms for finding either feasible or minimum-cost flows [2].

In our flow-based encodings, a consistency check will correspond to finding a feasible or minimum cost flow. To enforce DC, we therefore need an algorithm that, given a minimum cost flow of cost $w(f)$ and an edge e checks if an extra unit flow can be pushed (or removed) through the edge e and the cost of the resulting flow is less than or equal to a given threshold T. We use the residual graph to construct such an algorithm. Suppose we need to check if an extra unit flow can be pushed through an edge $e = (u,v)$. Let $e' = (u,v)$ be the corresponding arc in the residual graph. If $w(e) = 0, \forall e \in E$, then it is sufficient to compute strongly connected components (SCC) in the residual graph. An extra unit flow can be pushed through an edge e

[1] NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council.
[2] NICTA and UNSW, Sydney, Australia
[3] NICTA and UNSW, Sydney, Australia
[4] Ecole Polytechnique de Montreal, Montreal, Canada
[5] NICTA and UNSW, Sydney, Australia

iff both ends of the edge e' are in the same strongly connected component. If $w(e) \in \mathbb{Z}, \forall e \in E$, the shortest path p between v and u in the residual graph has to be computed. The minimal cost of pushing an extra unit flow through an edge e equals $w(f) + w(p) + w(e)$. If $w(f)+w(p)+w(e) > T$, then we cannot push an extra unit through e. Similarly, we can check if we can remove a unit flow through an edge.

3 The SEQUENCE Constraint

The SEQUENCE constraint was introduced by Beldiceanu and Contejean [5]. It constrains the number of values taken from a given set in any sequence of k variables. It is useful in staff rostering to specify, for example, that every employee has at least 2 days off in any 7 day period. Another application is sequencing cars along a production line (prob001 in CSPLib). It can specify, for example, that at most 1 in 3 cars along the production line has a sun-roof. The SEQUENCE constraint can be defined in terms of a conjunction of AMONG constraints. AMONG$(l, u, [X_1, \ldots, X_k], v)$ holds iff $l \leq |\{i | X_i \in v\}| \leq u$. That is, between l and u of the k variables take values in v. The AMONG constraint can be encoded by channelling into 0/1 variables using $Y_i \leftrightarrow (X_i \in v)$ and $l \leq \sum_{i=1}^{k} Y_i \leq u$. Since the constraint graph of this encoding is Berge-acyclic, this does not hinder propagation. Consequently, we will simplify notation and consider AMONG (and SEQUENCE) on 0/1 variables and $v = \{1\}$. If $l = 0$, AMONG is an ATMOST constraint. ATMOST is *monotone* since, given a support, we also have support for any larger value [6]. The SEQUENCE constraint is a conjunction of overlapping AMONG constraints. More precisely, SEQUENCE$(l, u, k, [X_1, \ldots, X_n], v)$ holds iff for $1 \leq i \leq n - k + 1$, AMONG$(l, u, [X_i, \ldots, X_{i+k-1}], v)$ holds. A sequence like $X_i, \ldots, X_{i+k-1}$ is a *window*. It is easy to see that this decomposition hinders propagation. If $l = 0$, SEQUENCE is an ATMOSTSEQ constraint. Decomposition in this case does not hinder propagation. Enforcing *DC* on the decomposition of an ATMOSTSEQ constraint is equivalent to enforcing *DC* on the ATMOSTSEQ constraint [6].

Several filtering algorithms exist for SEQUENCE and related constraints. Regin and Puget proposed a filtering algorithm for the Global Sequencing constraint (GSC) that combines a SEQUENCE and a global cardinality constraint (GCC) [17]. Beldiceanu and Carlsson suggested a greedy filtering algorithm for the CARDPATH constraint that can be used to propagate the SEQUENCE constraint, but this may hinder propagation [3]. Regin decomposed GSC into a set of variable disjoint AMONG and GCC constraints [16] but this decomposition also hinders propagation. Bessiere *et al.* [6] encoded SEQUENCE using a SLIDE constraint, and give a domain consistency propagator that runs in $O(nd^{k-1})$ time. van Hoeve *et al.* [13] proposed two filtering algorithms that establish domain consistency. The first is based on an encoding into a REGULAR constraint and runs in $O(n2^k)$ time, whilst the second is based on cumulative sums and runs in $O(n^3)$ time. Finally, Brand *et al.* [9] studied a number of different encodings of the SEQUENCE constraint. Their asymptotically fastest encoding is based on separation theory and enforces domain consistency in $O(n^2 \log n)$ time down the whole branch of a search tree. One of our contributions is to improve on this bound.

4 Flow-based Propagator for the SEQUENCE Constraint

We will convert the SEQUENCE constraint to a flow by means of a linear program (LP). We shall use SEQUENCE$(l, u, 3, [X_1, \ldots, X_6], v)$ as a running example. We can formulate this constraint simply and directly as an integer linear program:

$$\begin{aligned} l \leq \quad & X_1 + X_2 + X_3 \quad \leq u, \\ l \leq \quad & X_2 + X_3 + X_4 \quad \leq u, \\ l \leq \quad & X_3 + X_4 + X_5 \quad \leq u, \\ l \leq \quad & X_4 + X_5 + X_6 \quad \leq u \end{aligned}$$

where $X_i \in \{0, 1\}$. By introducing surplus/slack variables, Y_i and Z_i, we convert this to a set of equalities:

$$\begin{aligned} X_1 + X_2 + X_3 - Y_1 &= l, & X_1 + X_2 + X_3 + Z_1 &= u, \\ X_2 + X_3 + X_4 - Y_2 &= l, & X_2 + X_3 + X_4 + Z_2 &= u, \\ X_3 + X_4 + X_5 - Y_3 &= l, & X_3 + X_4 + X_5 + Z_3 &= u, \\ X_4 + X_5 + X_6 - Y_4 &= l, & X_4 + X_5 + X_6 + Z_3 &= u \end{aligned}$$

where $Y_i, Z_i \geq 0$. In matrix form, this is:

$$\begin{pmatrix} 1&1&1&0&0&0&-1&0&0&0&0&0&0&0 \\ 1&1&1&0&0&0&0&1&0&0&0&0&0&0 \\ 0&1&1&1&0&0&0&0&-1&0&0&0&0&0 \\ 0&1&1&1&0&0&0&0&0&1&0&0&0&0 \\ 0&0&1&1&1&0&0&0&0&0&-1&0&0&0 \\ 0&0&1&1&1&0&0&0&0&0&0&1&0&0 \\ 0&0&0&1&1&1&0&0&0&0&0&0&-1&0 \\ 0&0&0&1&1&1&0&0&0&0&0&0&0&1 \end{pmatrix} \begin{pmatrix} X_1 \\ \vdots \\ X_6 \\ Y_1 \\ Z_1 \\ \vdots \\ Y_4 \\ Z_4 \end{pmatrix} = \begin{pmatrix} l \\ u \\ l \\ u \\ l \\ u \\ l \\ u \end{pmatrix}$$

This matrix has the *consecutive ones* property for columns: each column has a block of consecutive 1's or -1's and the remaining elements are 0's. Consequently, we can apply the method of Veinott and Wagner [1] (also described in Application 9.6 of [2]) to simplify the problem. We create a zero last row and subtract the ith row from $i + 1$th row for $i = 1$ to $2n$. These operations do not change the set of solutions. This gives:

$$A\vec{X} = \vec{b},$$

where

$$A = \begin{pmatrix} 1&1&1&0&0&0&-1&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&1&1&0&0&0&0&0&0 \\ -1&0&0&1&0&0&0&-1&-1&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&1&1&0&0&0&0 \\ 0&-1&0&0&1&0&0&0&0&-1&-1&0&0&0 \\ 0&0&0&0&0&0&0&0&0&0&1&1&0&0 \\ 0&0&-1&0&0&1&0&0&0&0&0&-1&-1&0 \\ 0&0&0&0&0&0&0&0&0&0&0&0&1&1 \\ 0&0&0&-1&-1&-1&0&0&0&0&0&0&0&-1 \end{pmatrix},$$

$$\vec{X} = \left(X_1, \ldots, X_6, Y_1, Z_1, \ldots, Y_4, Z_4\right)^T,$$

$$\vec{b} = \left(l, u-l, l-u, u-l, l-u, u-l, l-u, u-l, -u\right)^T$$

This matrix has a single 1 and -1 in each column. Hence, it describes a network flow problem [2] on a graph $G = (V, E)$ (that is, it is a network matrix). Each row in the matrix corresponds to a node in V and each column corresponds to an edge in E. Down each column, there is a single row i equal to 1 and a single row j equal to -1 corresponding to an edge $(i, j) \in E$ in the graph. We include a source node s and a sink node t in V. Let b be the vector on the right hand side of the equation. If b_i is positive, then there is an edge $(s, i) \in E$ that carries exactly b_i amount of flow. If b_i is negative, there is an edge $(i, t) \in E$ that caries exactly $|b_i|$ amount of flow. The bounds on the variables, which are not expressed in the matrix, are represented as bounds on the capacity of the corresponding edges.

The graph for the set of equations in the example is given in Figure 1. A flow of value $4u - 3l$ in the graph corresponds to a solution. If a feasible flow sends a unit flow through the edge labeled with X_i then $X_i = 1$ in the solution; otherwise $X_i = 0$. Each even numbered

vertex $2i$ represents a window. The way the incoming flow is shared between y_j and z_j reflects how many variables X_i in the j'th window are equal to 1. Odd numbered vertices represent transitions from one window to the next (except for the first and last vertices, which represent transitions between a window and nothing). An incoming X edge represents the variable omitted in the transition to the next window, while an outgoing X edge represents the added variable.

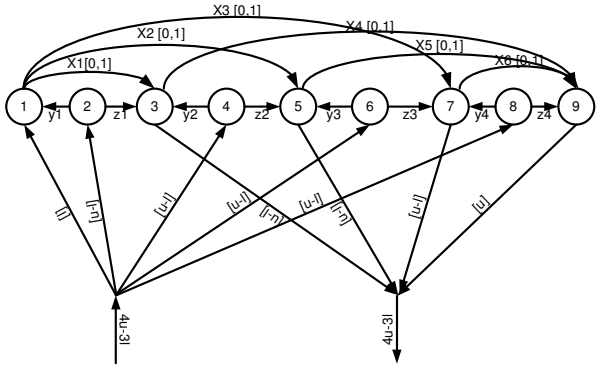


Figure 1. A flow graph for SEQUENCE$(l, u, 3, [X_1, \ldots, X_6], v)$

Theorem 1 *For any constraint* SEQUENCE$(l, u, k, [X_1, \ldots, X_n], v)$*, there is an equivalent network flow graph* $G = (V, E)$ *with* $5n - 4k + 5$ *edges,* $2n - 2k + 3 + 2$ *vertices, a maximum edge capacity of* u*, and an amount of flow to send equal to* $f = (n - k)(u - l) + u$*. There is a one-to-one correspondence between solutions of the constraint and feasible flows in the network.*

The time complexity of finding a maximum flow of value f is $O(|E|f)$ using the Ford-Fulkerson algorithm [10]. Faster algorithms exist for this problem. For example, Goldberg and Rao's algorithm finds a maximum flow in $O(min(|V|^{2/3}, |E|^{1/2})|E| \log(|V|^2/|E| + 2) \log C)$ time where C is the maximum capacity upper bound for an edge [12]. In our case, this gives $O(n^{3/2} \log n \log u)$ time complexity.

We follow Régin [14, 15] in the building of an incremental filtering algorithm from the network flow formulation. A feasible flow in the graph gives us a support for one value in each variable domain. Suppose $X_k = v$ is in the solution that corresponds to the feasible flow where v is either zero or one. To obtain a support for $X_k = 1 - v$, we find the SCC of the residual graph and check if both ends of the edge labeled with X_k are in the same strongly connected component. If so, $X_k = 1 - v$ has a support; otherwise $1 - v$ can be removed from the domain of X_k. Strongly connected components can be found in $O(|E|) = O(n)$, because the number of edges in the flow graph for the SEQUENCE constraint is linear in n by Theorem 1.

The total time complexity for initially enforcing DC is $O(n((n - k)(u - l) + u))$ if we use the Ford-Fulkerson algorithm or $O(n^{3/2} \log n \log u)$ if we use Goldberg and Rao's algorithm.

Still following Régin [14, 15], one can make the algorithm incremental. Suppose during search X_i is fixed to value v. If the last computed flow was a support for $X_i = v$, then there is no need to recompute the flow. We simply need to recompute the SCC in the new residual graph and enforce DC in $O(n)$ time. If the last computed flow is not a support for $X_i = v$, we can find a cycle in the residual graph containing the edge associated to X_i in $O(n)$ time. By pushing a unit of flow over this cycle, we obtain a flow that is a support for $X_i = v$. Enforcing DC can be done in $O(n)$ after computing the SCC. Consequently, there is an incremental cost of $O(n)$ when a variable is fixed, and the cost of enforcing DC down a branch of the search tree is $O(n^2)$.

5 Soft SEQUENCE Constraint

Soft forms of the SEQUENCE constraint may be useful in practice. The ROADEF 2005 challenge [19], which was proposed and sponsored by Renault, put forwards a violation measure for the SEQUENCE constraint which takes into account by how much each AMONG constraint is violated. We therefore consider the soft global constraint, SOFTSEQUENCE$(l, u, k, T, [X_1, \ldots, X_n], v)$. This holds iff:

$$T \quad \geq \quad \sum_{i=1}^{n-k+1} \max(l - S_i, S_i - u, 0), \tag{1}$$

where $S_i = \sum_{j=0}^{k-1} (X_{i+j} \in v)$

As before, we can simplify notation and consider SOFTSEQUENCE on 0/1 variables and $v = \{1\}$.

We again convert to a flow problem by means of a linear program, but this time with an objective function. Consider SOFTSEQUENCE$(l, u, 3, T, [X_1, \ldots, X_6], v)$. We introduce variables, Q_i and P_i to represent the penalties that may arise from violating lower and upper bounds respectively. We can then express this SOFTSEQUENCE constraint as follows. The objective function gives a lower bound on T.

$$\text{Minimize} \sum_{i=1}^{4} (P_i + Q_i) \text{ subject to :}$$

$$X_1 + X_2 + X_3 - Y_1 + Q_1 = l,\ X_1 + X_2 + X_3 + Z_1 - P_1 = u,$$
$$X_2 + X_3 + X_4 - Y_2 + Q_2 = l,\ X_2 + X_3 + X_4 + Z_2 - P_2 = u,$$
$$X_3 + X_4 + X_5 - Y_3 + Q_3 = l,\ X_3 + X_4 + X_5 + Z_3 - P_3 = u,$$
$$X_4 + X_5 + X_6 - Y_4 + Q_4 = l,\ X_4 + X_5 + X_6 + Z_3 - P_4 = u$$

where Y_i, Z_i, P_i and Q_i are non-negative. In matrix form, this is:

$$\text{Minimize } \textstyle\sum_{i=1}^{4} (P_i + Q_i) \quad \text{subject to:}$$

$$A\vec{X} = \vec{b},$$

where

$$A = \begin{pmatrix}
1&1&1&0&0&0&-1&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0\\
1&1&1&0&0&0&0&1&0&0&0&0&0&0&0&-1&0&0&0&0&0&0&0\\
0&1&1&1&0&0&0&0&-1&0&0&0&0&0&0&0&1&0&0&0&0&0&0\\
0&1&1&1&0&0&0&0&0&1&0&0&0&0&0&0&0&-1&0&0&0&0&0\\
0&0&1&1&1&0&0&0&0&0&-1&0&0&0&0&0&0&0&1&0&0&0&0\\
0&0&1&1&1&0&0&0&0&0&0&1&0&0&0&0&0&0&0&-1&0&0&0\\
0&0&0&1&1&1&0&0&0&0&0&0&-1&0&0&0&0&0&0&0&1&0&0\\
0&0&0&1&1&1&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&-1&0
\end{pmatrix}$$

$$\vec{X} = \left(X_1, \ldots, X_6, Y_1, Z_1, \ldots, Y_4, Z_4, Q_1, P_1, \ldots, Q_4, P_4\right)^T,$$

$$\vec{b} = \left(l, u, l, u, l, u, l, u\right)^T$$

If we transform the matrix as before, we get a minimum cost network flow problem.

$$\text{Minimize } \textstyle\sum_{i=1}^{4} (P_i + Q_i) \quad \text{subject to:}$$

$$A\vec{X} = \vec{b},$$

where

$$\text{A} = \begin{bmatrix} A1 \mid & A2 \mid & -A2 \end{bmatrix}$$

$$A1 = \begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & -1 & -1 \end{pmatrix}$$

$$A2 = \begin{pmatrix} -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \end{pmatrix}$$

$$\vec{X} = \left(X_1, \ldots, X_6, Y_1, Z_1, \ldots, Y_4, Z_4, Q_1, P_1, \ldots, Q_4, P_4\right)^T$$

$$\vec{b} = \left(l, u-l, l-u, u-l, l-u, u-l, l-u, u-l, -u\right)^T$$

The flow graph $G = (V, E)$ for the transformed system is presented in Figure 2. Dashed edges have cost 1, while other edges have cost 0. The minimal cost flow in the graph corresponds to a minimal cost solution to the system of equations.

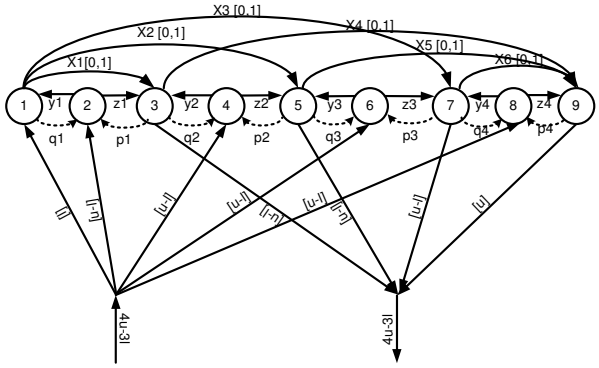


Figure 2. A flow graph for SOFTSEQUENCE$(l, u, 3, t, [X_1, \ldots, X_6])$

Theorem 2 *For any constraint* SOFTSEQUENCE $(l, u, k, t, [X_1, \ldots, X_n], v)$*, there is an equivalent network flow graph. There is a one-to-one correspondence between solutions of the constraint and feasible flows of cost less than or equal to* t*.*

Thus, if the minimal cost flow is greater than $\max(dom(T))$, then the SOFTSEQUENCE constraint is inconsistent. The minimal cost flow can be found in $O(|V||E| \log \log U \log |V|C) = O(n^2 \log n \log \log u)$ time [2]. Consider the edge (u, v) in the residual graph associated to variable X_i and let $k_{(u,v)}$ be its residual cost. If the flow corresponds to an assignment with $X_i = 0$, pushing a unit of flow on (u, v) results in a solution with $X_i = 1$. Symmetrically, if the flow corresponds to an assignment with $X_i = 1$, pushing a unit of flow on (u, v) results in a solution with $X_i = 0$. If the shortest path in the residual graph between v and u is $k_{(v,u)}$, then the shortest cycle that contains (u, v) has length $k_{(u,v)} + k_{(v,u)}$. Pushing a unit of flow through this cycle results in a flow of cost $c + k_{(u,v)} + k_{(v,u)}$ which is the minimum-cost flow that contains the edge (u, v). If $c + k_{(u,v)} + k_{(v,u)} > \max(dom(T))$, then no flows containing the edge (u, v) exist with a cost smaller or equal to $\max(dom(T))$. The variable X_i must therefore be fixed to the value taken in the current flow. Following Equation 1, the cost of the variable T must be no smaller than the cost of the solution. To enforce bounds consistency on the cost variable, we increase the lower bound of $dom(T)$ to the cost of the minimum flow in the graph G.

To enforce DC on the X variables efficiently we can use an all pairs shortest path algorithm on the residual graph. This takes $O(n^2 \log n)$ time using Johnson's algorithm [10]. This gives an $O(n^2 \log n \log \log u)$ time complexity to enforce DC on the SOFTSEQUENCE constraint. The penalty variables used for SOFTSEQUENCE arise directly out of the problem description and occur naturally in the LP formulation. We could also view them as arising through the methodology of [20], where edges with costs are added to the network graph for the hard constraint to represent the softened constraint.

5.1 Soft ATMOSTSEQ Constraint

In many cases, we have only upper bounds and not lower bounds on the frequency of the occurrence of values (i.e. $l = 0$). For instance, this is the case in car sequencing problems. This can be used to simplify propagation. For example, there is a simple propagator to enforce DC on the soft ATMOSTSEQ constraint in just $O(n^2 k)$ time down a branch of the search tree. Consider the assignment which assigns each X_i the smallest value in its domain. Due to monotonicity of the ATMOSTSEQ constraint any other solution X'_i will be greater or equal to this minimal assignment: $X_i \leq X'_i$, $i = 1, \ldots, n$. The violation measure is a monotonically non-decreasing function of the X_i. Consequently, the violation cost for any other solution is greater or equal to the violation cost of this minimal assignment. Hence, if the violation cost for the minimal assignment is greater than the upper bound on the cost variable then the constraint is inconsistent. To enforce DC on soft ATMOSTSEQ, we can use the failed literal test. If a value is pruned from the domain of X_i, then it takes $O(k)$ time to update the cost value of the minimal assignment and $O(nk)$ time to perform the failed literal test for n Boolean variables. Hence, the total time complexity is $O(n^2 k)$ down a branch of the search tree.

6 Generalized SEQUENCE Constraint

To model real world problems, we may want to have different size or positioned windows. For example, the window size in a rostering problem may depend on whether it includes a weekend or not. An extension of the SEQUENCE constraint proposed in [13] is that each AMONG constraint can have different parameters (start position, l, u, and k). More precisely, GEN-SEQUENCE$(\vec{p_1}, \ldots, \vec{p_m}, [X_1, X_2, \ldots, X_n], v)$ holds iff AMONG$(l_i, u_i, k_i, [X_{s_i}, \ldots, X_{s_i+k_i-1}], v)$ for $1 \leq i \leq m$ where $\vec{p_i} = \langle l_i, u_i, k_i, s_i \rangle$. Whilst the methods in Section 4 easily extend to allow different bounds l and u for each window, dealing with different windows is more difficult. In general, the matrix now does not have the consecutive ones property. It may be possible to reorder the windows to achieve the consecutive ones property. If such a re-ordering exists, it can be found and performed in $O(m+n+r)$ time, where r is the number of non-zero entries in the matrix [8]. Even when re-ordering cannot achieve the consecutive ones property there may, nevertheless, be an equivalent network matrix. Bixby

and Cunningham [7] give a procedure[6] to find an equivalent network matrix, when it exists, in $O(mr)$ time. In these cases, the method in Section 4 can be applied to propagate the GEN-SEQUENCE constraint in $O(n^2)$ time down the branch of a search tree.

Not all GEN-SEQUENCE constraints can be expressed as network flows. Consider the GEN-SEQUENCE constraint with $n = 5$, identical upper and lower bounds (l and u), and 4 windows: [1,5], [2,4], [3,5], and [1,3]. We can express it as an integer linear program:

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ -1 & -1 & -1 & -1 & -1 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & -1 & -1 & -1 & 0 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & -1 & -1 & -1 \\ 1 & 1 & 1 & 0 & 0 \\ -1 & -1 & -1 & 0 & 0 \end{pmatrix} \begin{pmatrix} X_1 \\ X_2 \\ X_3 \\ X_4 \\ X_5 \end{pmatrix} \geq \begin{pmatrix} l \\ -u \\ l \\ -u \\ l \\ -u \\ l \\ -u \end{pmatrix} \tag{2}$$

Applying the test described in Section 20.1 of [18] to Example 2, we find that the matrix of this problem is not equivalent to any network matrix.

However, all GEN-SEQUENCE constraint matrices satisfy a weaker property: total unimodularity. A matrix is *totally unimodular* iff every square non-singular submatrix has a determinant of $+1$ or -1. The advantage of this property is that any totally unimodular system of inequalities with integral constants is solvable in $\mathbb{Z}$ iff it is solvable in $\mathbb{R}$.

Theorem 3 *The matrix of the inequalities associated with* GEN-SEQUENCE *constraint is totally unimodular.*

In practice, only integral values for the bounds l_i and u_i are used. Thus the consistency of a GEN-SEQUENCE constraint can be determined via linear programming techniques in $O(n^{3.5} \log u)$ time. Using the failed literal test, we can enforce DC at a cost of $O(n^{5.5} \log u)$ down the branch of a search tree for any GEN-SEQUENCE constraint. This is too expensive to be practical. We can, instead, exploit the fact that the matrix for each GEN-SEQUENCE constraint has the consecutive ones property *for rows* (before the introduction of slack/surplus variables). Corresponding to the row transformation for matrices with consecutive ones for columns is a change-of-variables transformation into variable $S_j = \sum_{i=1}^{j} X_i$ for matrices with consecutive ones for rows. This gives the dual of a network matrix. This is the basis of an encoding of SEQUENCE in [9] (denoted there CD). Consequently that encoding extends to GEN-SEQUENCE. Adapting the analysis in [9] to GEN-SEQUENCE, we can enforce DC in $O(nm + n^2 \log n)$ time down the branch of a search tree.

In summary, for a compilation cost of $O(mr)$, we can enforce DC on a GEN-SEQUENCE constraint in $O(n^2)$ down the branch of a search tree, when it has a flow representation, and in $O(nm + n^2 \log n)$ when it does not.

7 SLIDINGSUM Constraint

The SLIDINGSUM constraint [4] is a generalization of the SEQUENCE constraint from Boolean to integer variables, which we extend to allow arbitrary windows. SLIDINGSUM $([X_1, \ldots, X_n], [\vec{p_1}, \ldots, \vec{p_m}])$ holds iff $l_i \leq \sum_{j=s_i}^{s_i+k_i-1} X_i \leq u_i$ holds where $\vec{p_i} = \langle l_i, u_i, k_i, s_i \rangle$ is, as with the generalized SEQUENCE, a window. The constraint can be expressed as a linear program $\mathcal{P}$ called the *primal* where W is a matrix encoding the inequalities and the bounds on each variable are given by $a_i \leq X_i \leq b_i$. Since the constraint represents a satisfaction problem, we minimize the constant 0.

$$\left.\begin{array}{c} \min\ 0 \\ \begin{bmatrix} W \\ -W \\ I \\ -I \end{bmatrix} X \geq \begin{bmatrix} l \\ -u \\ a \\ -b \end{bmatrix} \end{array}\right\} \mathcal{P} \tag{3}$$

The *dual* $\mathcal{D}$ is however an optimization problem.

$$\left.\begin{array}{c} \min \begin{bmatrix} -l & u & -a & b \end{bmatrix} Y \\ \begin{bmatrix} W^T & -W^T & I & -I \end{bmatrix} Y = 0 \\ Y \geq 0 \end{array}\right\} \mathcal{D} \tag{4}$$

Von Neumann's Strong Duality Theorem states that if the primal and the dual problems are feasible, then they have the same objective value. Moreover, if the primal is unsatisfiable, the dual is unbounded. The SLIDINGSUM constraint is thus satisfiable if the objective function of the dual problem is zero. It is unsatisfiable if it tends to negative infinity.

Note that the matrix W^T has the consecutive ones property on the columns. The dual problem can thus be converted to a network flow using the same transformation as with the SEQUENCE constraint. Consider the dual LP of our running example 2:

$$\begin{array}{c} \text{Minimize} \\ -\sum_{i=1}^{4} l_i Y_i + \sum_{i=1}^{4} u_i Y_{4+i} - \sum_{i=1}^{5} a_i Y_{8+i} + \sum_{i=1}^{5} b_i Y_{13+i} \\ \text{subject to:} \end{array}$$

$$A\vec{Y} = \vec{b},$$

where

$$A = \begin{pmatrix} 1 & 0 & 0 & 1 & -1 & 0 & 0 & -1 & 1 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & -1 & -1 & 0 & -1 & 0 & 1 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & -1 & 0 & 0 \\ 1 & 1 & 1 & 0 & -1 & -1 & -1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & -1 & 0 \\ 1 & 0 & 1 & 0 & -1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & -1 \end{pmatrix},$$

$$\vec{Y} = \left(Y_1, \ldots, Y_{18}\right)^T,$$

$$\vec{b} = \left(0, \ldots, 0\right)^T.$$

Our usual transformation will turn this into a network flow problem:

$$\begin{array}{c} \text{Minimize} \\ -\sum_{i=1}^{4} l_i Y_i + \sum_{i=1}^{4} u_i Y_{4+i} - \sum_{i=1}^{5} a_i Y_{8+i} + \sum_{i=1}^{5} b_i Y_{13+i} \\ \text{subject to} \end{array}$$

$$A\vec{Y} = \vec{0},$$

where

$$A = \begin{pmatrix} 1 & 0 & 0 & 1 & -1 & 0 & 0 & -1 & 1 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & -1 & 0 & 0 & -1 & 1 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & -1 & 0 & 0 & -1 & 1 & 0 & 0 & 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 & 0 & 0 & 1 & 0 & 0 & -1 & 1 & 0 & 0 & 0 & 1 & -1 & 0 \\ 0 & -1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & -1 & 1 & 0 & 0 & 0 & 1 & -1 \\ -1 & 0 & -1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 1 \end{pmatrix},$$

$$\vec{Y} = \left(Y_1, \ldots, Y_{18}\right)^T.$$

The flow associated with this example is given in Figure 3. There are $n + 1$ nodes labelled from 1 to $n + 1$ where node i is connected

[6] Another procedure is given in [18].

to node $i+1$ with an edge of cost $-a_i$ and node $i+1$ is connected to node i with an edge of cost b_i. For each window $\vec{p_i}$, we have an edge from s_i to $s_i + k_i$ with cost $-l_i$ and an edge from $s_i + k_i$ to s_i with cost u_i. All nodes have a null supply and a null demand. A flow is therefore simply a circulation i.e., an amount of flow pushed on the cycles of the graph.

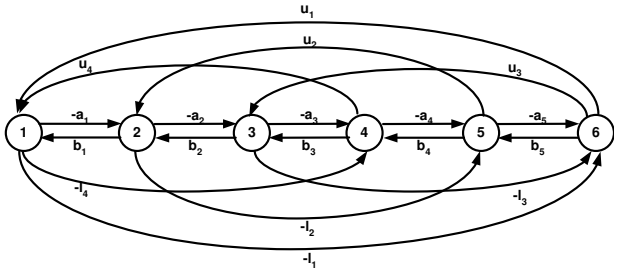


Figure 3. Network flow associated to the SLIDINGSUM constraint posted on the running example.

Theorem 4 *The* SLIDINGSUM *constraint is satisfiable if and only there are no negative cycles in the flow graph associated with the dual linear program.*

Proof: If there is a negative cycle in the graph, then we can push an infinite amount of flow resulting in a cost infinitely small. Hence the dual problem is unbounded, and the primal is unsatisfiable. Suppose that there are no negative cycles in the graph. Pushing any amount of flow over a cycle of positive cost results in a flow of cost greater than zero. Such a flow is not optimal since the null flow has a smaller objective value. Pushing any amount of flow over a null cycle does not change the objective value. Therefore the null flow is an optimal solution and since this solution is bounded, then the primal is satisfiable. Note that the objective value of the dual (zero) is in this case equal to the objective value of the primal. $\diamond$

The flow graph has $O(n)$ nodes and $O(n+m)$ edges. Testing whether there is a negative cycle takes $O(n^2 + nm)$ time using the Bellman-Ford algorithm. We can use this consistency test to construct an efficient bounds consistency propagator. We find for each variable X_i the smallest (largest) value in its domain such that assigning this value to X_i does not create a negative cycle. We compute the shortest path between all pairs of nodes. Johnson's algorithm solves the all-pair shortest path problem in $O(|V|^2 \log |V| + |V||E|)$ time which in our case gives $O(n^2 \log n + nm)$ time. Suppose that the shortest path between i and $i+1$ has length $s(i, i+1)$, then for the constraint to be satisfiable, we need $b_i + s(i, i+1) \geq 0$. Since b_i is a value potentially taken by X_i, we need to have $X_i \geq -s(i, i+1)$. We therefore assign $\min(dom(X_i)) \leftarrow \max(\min(dom(X_i)), -s(i, i+1))$. Similarly, let the length of the shortest path between $i+1$ and i be $s(i+1, i)$. For the constraint to be satisfiable, we need $s(i+1, i) - a_i \geq 0$. Since a_i is a value potentially taken by X_i, we have $X_i \leq s(i+1, i)$. We assign $\max(X_i) \leftarrow \min(\max(X_i), s(i+1, i))$. It is not hard to prove this is sound and complete, removing all values that cause negative cycles. Following [9], we can make the propagator incremental using the algorithm by Cotton and Maler [11] to maintain the shortest path between $|P|$ pairs of nodes in $O(|E| + |V| \log |V| + |P|)$ time upon edge reduction. Each time a lower bound a_i is increased or an upper bound b_i is decreased, the shortest paths can be recomputed in $O(m + n \log n)$ time.

8 Soft SLIDINGSUM Constraint

The soft SLIDINGSUM constraint is an extension of the SLIDINGSUM constraint. The soft SLIDINGSUM $([X_1, \ldots, X_n], [\vec{p_1}, \ldots, \vec{p_m}], T)$ introduces a violation variable T and is defined as follow.

$$T \geq \sum_{i=1}^{m} \max(l_i - \sum_{j=s_i}^{s_i+k_i-1} X_j, \sum_{j=s_i}^{s_i+k_i-1} X_j - u_i, 0) \quad (5)$$

To express the soft SLIDINGSUM constraint as a linear program, we introduce penalty variables for each inequality associated to the hard SLIDINGSUM, namely, Q_i and P_i, $i = 1, \ldots, m$ and minimize the sum of penalty variables:

$$\min \sum_{i=1}^{m} Q_i + P_i \quad (6)$$

$$\sum_{j \in s_i}^{s_i+k_i-1} X_j + Q_i \geq l_i \qquad \forall\, 1 \leq i \leq m \quad (7)$$

$$\sum_{j \in s_i}^{s_i+k_i-1} -X_j + P_i \geq -u_i \qquad \forall\, 1 \leq i \leq m \quad (8)$$

$$X_i \geq a_i,\ -X_i \geq -b_i,\ Q_i \geq 0,\ P_i \geq 0 \quad (9)$$

Rewriting system (6)– (9) in the matrix form, we obtain primal linear program $\mathcal{P}$:

$$\left.\begin{array}{l} \min e^T Q + e^T P \\ \begin{bmatrix} W & I_m & 0 \\ -W & 0 & I_m \\ I_n & 0 & 0 \\ -I_n & 0 & 0 \\ 0 & I_m & 0 \\ 0 & 0 & I_m \end{bmatrix} \begin{bmatrix} X \\ Q \\ P \end{bmatrix} \geq \begin{bmatrix} l \\ -u \\ a \\ -b \\ 0 \\ 0 \end{bmatrix} \end{array}\right\} \mathcal{P} \quad (10)$$

where I_n is the $n \times n$ identity matrix, l and u are the vectors containing the m values l_i and u_i, a and b are the vectors containing the n lower and upper bounds a_i and b_i, and e is the vector of dimension m with all components set to one.

The *dual* problem $\mathcal{D}$ corresponding to the primal problem $\mathcal{P}$ (system (10)) is

$$\left.\begin{array}{l} \min \begin{bmatrix} -l & u & -a & b & 0 & 0 \end{bmatrix}^T Y \\ \begin{bmatrix} W^T & -W^T & I_n & -I_n & 0 & 0 \\ I_m & 0 & 0 & 0 & I_m & 0 \\ 0 & I_m & 0 & 0 & 0 & I_m \end{bmatrix} Y = \begin{bmatrix} 0 \\ e \\ e \end{bmatrix} \\ Y \geq 0 \end{array}\right\} \mathcal{D} \quad (11)$$

where Y is a vector of $4n + 2m$ *dual variables*.

The dual problem $\mathcal{D}$ can be transformed using row operations to obtain the consecutive ones property on the columns of the matrix. Note that W^T already has the consecutive ones property. For each of the first m columns, one needs to obtain ones between the last entry in W^T set to one and the identity matrix under W^T. This is done

by selecting the row in the identity matrix whose corresponding column is set to one and adding this row to every row above until the consecutive ones property is reached on this column. The principle applies to the m following columns except that the last m equations are negated to obtain columns with negative ones. The following $2n$ columns already had the consecutive ones property and remain unchanged during the transformation. The last $2m$ columns are modified but still satisfy the consecutive ones property. Using the same technique for the SEQUENCE constraint, we obtain a system that can be solved using a network flow algorithm.

Theorem 5 *There is a one-to-one correspondence between solutions of the soft* SLIDINGSUM *constraint and feasible flows of cost less than or equal to the upper bound of* T.

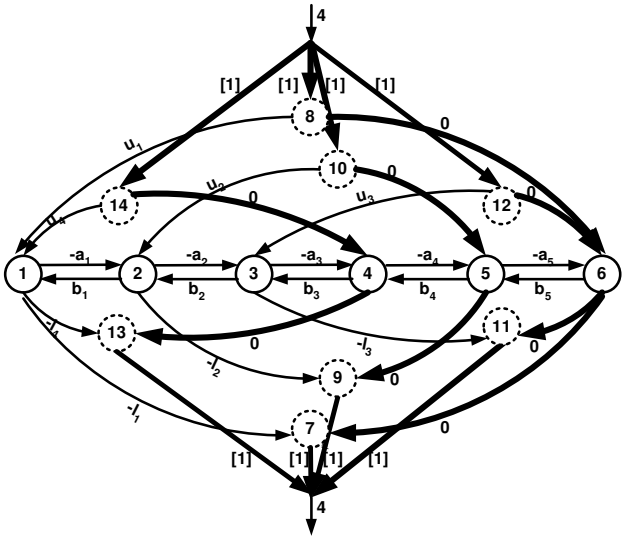


Figure 4. Network flow associated with the soft SLIDINGSUM constraint posted on the running example. The edge capacities are written in square brackets [] to differentiate them from the edge costs. Bold edges show a possible flow in the network.

Figure 8 shows the flow graph for the soft version of the running example in Section 7. Note that the flow graphs for the hard and soft SLIDINGSUM constraints have a very similar structure. Consider the flow graph for the SLIDINGSUM constraint in the running example (Figure 3). It includes arcs that correspond to original variables X and are labeled $-a_i$ or b_i, $i = 1, \ldots, n$ and arcs that correspond to linear inequalities and are labeled $-l_i$ or u_i, $i = 1, \ldots, m$. The flow graph for the soft SLIDINGSUM constraint contains the same arcs for variables, however, each of the inequality arcs is split into two arcs by introducing a node with unit demand or supply[7]. This leads to a difference between the two flow graphs: the flow graph for the SLIDINGSUM constraint has zero flow circulation, while the flow graph for the soft SLIDINGSUM constraint contains a flow of value m. Note that the capacity of each edge connecting either the source or the sink to other nodes is exactly 1. However, the capacities of the other edges in the flow network are not bounded. Therefore, these edges can carry several units of flow in a feasible minimum cost flow, which makes a flow-based BC propagator for the soft SLIDINGSUM constraint more computationally expensive compared to the hard case. The flow graph for the soft SLIDINGSUM constraint has $O(n + m)$ nodes and $O(n + m)$ edges. The minimal cost flow can be found in $O(|V| \log |E|(|E| + |V| \log |V|) = O((n + m)^2 \log^2(n + m))$ time [2]. The BC filtering algorithm for the soft SLIDINGSUM constraint works exactly the same as for the soft SEQUENCEconstraint (Section 5), except that finding all pairs of shortest paths is replaced with finding all pairs of minimal cost flows. Hence, the total time complexity of the flow-based BC filtering algorithm is $O(n(n + m)^2 \log^2(n + m))$.

[7] Note that, in contrast to the soft SEQUENCE constraint, this flow graph is not obtained by the methodology of [20].

9 Cyclic SEQUENCE constraint

In rostering problems, we may wish to produce a cyclic schedule which can be repeated, say, every four weeks. We therefore consider a cyclic version of the SEQUENCE constraint. More precisely, CYCLICSEQUENCE$(l, u, k, [X_1, \ldots, X_n], v)$ ensures that between l and u variables in X_i to $X_{1+(i+k-1 \bmod n)}$ takes values in the set v for $1 \leq i \leq n$.

The cyclic SEQUENCE constraint can be expressed with a linear program. Consider, for example, the primal linear system for CYCLICSEQUENCE$(l, u, 2, [X_1, \ldots, X_3], v)$.

$$\begin{pmatrix} 1 & 1 & 0 \\ -1 & -1 & 0 \\ 0 & 1 & 1 \\ 0 & -1 & -1 \\ 1 & 0 & 1 \\ -1 & 0 & -1 \end{pmatrix} \begin{pmatrix} X_1 \\ X_2 \\ X_3 \end{pmatrix} \geq \begin{pmatrix} l \\ -u \\ l \\ -u \\ l \\ -u \end{pmatrix} \tag{12}$$

Unfortunately, the matrix at the left-hand side of system (12) is not totally unimodular, because it contains a submatrix with the determinant equal to 2.

$$\begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \end{pmatrix}$$

Hence, methods employing a network flow or its dual, as in Sections 4 and 6, are not directly applicable to Cyclic SEQUENCE.

10 Experimental Results

To evaluate performance of our filtering algorithms we carried out a series of experiments on random problems. The experimental setup is similar to that in [9]. The first set of experiments compares performance of the flow-based propagator FB on single instance of the SEQUENCE constraint against the $HPRS$ propagator (the third propagator in [13]), the CS encoding of [9], and the AMONG decomposition AD of the SEQUENCE constraint. The second set of experiments compares the flow-based propagator FB_S for the SOFTSEQUENCE constraint and its decomposition into soft AMONG constraints. Experiments were run with ILOG 6.1 on an Intel Xeon 4 CPU, 2.0 Ghz, 4G RAM. Boost graph library version 1.34.1 was used to implement the flow-based algorithms.

10.1 The SEQUENCE constraint

For each possible combination of $n \in \{500, 1000, 2000, 3000, 4000, 5000\}$, $k \in \{5, 15, 50\}$, $\Delta = u - l \in \{1, 5\}$, we generated twenty instances with random lower bounds in the interval $(0, k - \Delta)$. We used random value and variable ordering and a time out of 300 sec. We used

the Ford-Fulkerson algorithm to find a maximum flow. Results for different values of Δ are presented in Tables 1- 2 and Figure 5. First of all, we notice that the CS encoding is the best on hard instances ($\Delta = 1$) and the AD decomposition is the fastest on easy instances ($\Delta = 5$). This result was first observed in [9]. The FB propagator is not the fastest one but has the most robust performance. It is sensitive only to the value of n and not to other parameters, like the length of the window(k) or hardness of the problem(Δ). As can be seen from Figure 5, the FB propagator scales better than the other propagators with the size of the problem. It appears to grow linearly with the number of variables, while the $HPRS$ propagator displays quadratic growth.

n	k	AD	CS	$HPRS$	FB
500	7	8 / 2.13	**20 / 0.13**	**20** / 0.35	**20** / 0.30
	15	6 / 0.01	**20 / 0.09**	**20** / 0.30	**20** / 0.29
	50	2 / 0.02	**20 / 0.07**	**20** / 0.26	**20** / 0.28
1000	7	4 / 0.01	**20 / 0.71**	**20** / 2.36	**20** / 1.18
	15	2 / 0.59	**20 / 0.38**	**20** / 2.06	**20** / 1.17
	50	1 /0	**20 / 0.28**	**20** / 1.48	**20** / 1.14
2000	7	4 / 0.04	**20 / 4.25**	**20** / 18.52	**20** / 4.76
	15	0 /0	**20 / 1.84**	**20** / 15.19	**20** / 4.56
	50	1 /0	**20 / 1.16**	**20** / 13.24	**20** / 4.42
3000	7	3 / 0.07	**20** / 15.14	**20** / 64.04	**20 / 10.44**
	15	1 /0	**20 / 5.49**	**20** / 51.04	**20** / 11.90
	50	0 /0	**20 / 2.61**	**20** / 35.48	**20** / 10.12
4000	7	3 / 0.12	**20** / 30.87	**20** / 132.73	**20 / 23.25**
	15	0 /0	**20 / 14.44**	**20** / 123.60	**20** / 18.61
	50	1 /0	**20 / 4.78**	**20** / 93.98	**20** / 18.97
5000	7	1 /0	**20** / 64.05	15 / 262.17	**20 / 36.09**
	15	0 /0	**20 / 24.46**	17 / 211.17	**20** / 34.59
	50	0 /0	**20 / 8.24**	19 / 146.63	**20** / 31.66
TOTALS					
solved/total		37 /360	**360** /360	351 /360	**360** /360
avg tm for solved		0.517	**9.943**	60.973	11.874
avg bt for solved		17761	429	0	0

Table 1. Randomly generated instances with a single SEQUENCE constraint and $\Delta = 1$. Number of instances solved in 300 sec / average time to solve.

n	k	AD	CS	$HPRS$	FB
500	7	**20 / 0.01**	**20** / 0.58	**20** / 0.15	**20** / 0.44
	15	**20 / 0.01**	**20** / 0.69	**20** / 0.25	**20** / 0.44
	50	18 / 0.02	**20 / 0.20**	**20** / 0.37	**20** / 0.42
1000	7	**20 / 0.03**	**20** / 4.33	**20** / 0.99	**20** / 1.70
	15	**20 / 0.03**	**20** / 4.68	**20** / 1.83	**20** / 1.70
	50	10 / 0.05	**20 / 1.24**	**20** / 2.73	**20** / 1.69
2000	7	**20 / 0.07**	**20** / 32.41	**20** / 7.19	**20** / 6.62
	15	**20 / 0.07**	**20** / 39.71	**20** / 14.89	**20** / 6.63
	50	5 / 5.19	**20** / 9.52	**20** / 13.71	**20 / 6.94**
3000	7	**20 / 0.14**	**20** / 104.68	**20** / 23.85	**20** / 14.96
	15	**20 / 0.16**	**20** / 125.11	**20** / 44.67	**20** / 15.21
	50	5 / 0.29	**20** / 22.73	**20** / 66.61	**20 / 14.61**
4000	7	**20 / 0.25**	17 / 251.56	**20** / 55.70	**20** / 29.34
	15	**20 / 0.22**	5 / 179.41	**20** / 112.99	**20** / 26.99
	50	9 / 0.34	**20** / 50.52	17 / 141.25	**20 / 26.67**
5000	7	**20 / 0.36**	0 /0	**20** / 109.18	**20** / 46.42
	15	**20 / 0.36**	6 / 160.99	17 / 215.97	**20** / 45.97
	50	9 / 0.48	**20** / 108.34	11 / 210.53	**20 / 44.88**
TOTALS					
solved/total		296 /360	308 /360	345 /360	**360** /360
avg tm for solved		0.236	52.708	50.698	**16.200**
avg bt for solved		888	1053	0	**0**

Table 2. Randomly generated instances with a single SEQUENCE constraint and $\Delta = 5$. Number of instances solved in 300 sec / average time to solve.

		$\Delta = 1$		$\Delta = 5$	
n	k	AD_S	FB_S	AD_S	FB_S
50	7	6 / 19.30	**7 / 27.91**	**20 / 0.01**	**20** / 2.17
	15	8 / 36.07	**13 / 20.41**	**11 / 49.49**	10 / 30.51
	25	6 / 0.73	**10 / 23.27**	**10 / 6.40**	**10** / 7.41
100	7	1 /0	**3 / 7.56**	**19 / 10.50**	18 / 16.51
	15	0 /0	**5 / 6.90**	**3 / 0.01**	**3** / 7.20
	25	0 /0	**5 / 4.96**	**5 / 19.07**	**5** / 23.99
TOTALS					
solved/total		21 /120	**43** /120	**68** /120	66 /120
avg tm for solved		19.463	**18.034**	**13.286**	13.051
avg bt for solved		245245	**343**	**147434**	128

Table 3. Randomly generated instances with 4 soft SEQUENCEs. Number of instances solved in 300 sec / average time to solve.

10.2 The Soft SEQUENCE constraint

We evaluated performance of the soft SEQUENCE constraint on random problems. For each possible combination of $n \in \{50, 100\}$, $k \in \{5, 15, 25\}$, $\Delta = \{1, 5\}$ and $m \in \{4\}$ (where m is the number of SEQUENCE constraints), we generated twenty random instances. All variables had domains of size 5. An instance was obtained by selecting random lower bounds in the interval $(0, k - \Delta)$. We excluded instances where $\sum_{i=1}^{m} l_i \geq k$ to avoid unsatisfiable instances. We used a random variable and value ordering, and a time-out of 300 sec. All SEQUENCE constraints were enforced on disjoint sets of cardinality one. Instances with this set of parameters are hard instances for SEQUENCE propagators [9]. To relax these instances, we allow to violate the SEQUENCE constraint with a cost that has to be less than or equal to 15% of the length of the sequence. Experimental results are presented in Table 3. As can be seen from the table, the FB_S algorithms is competitive with the decomposition into soft AMONG constraints on easy problems and outperforms the decomposition on hard problems.

We observed that the FB_S propagator is very slow for the soft SEQUENCE constraint. Note that the number of backtracks of FB_S is three order of magnitude smaller compared to AD_S. We profiled the algorithm and found that it spends most of the time performing the all pairs shortest path algorithm. Unfortunately, this is difficult to compute incrementally because the residual graph can be different on every invocation of the propagator.

11 Conclusion

We have proposed new filtering algorithms for the SEQUENCE constraint and several extensions including the soft SEQUENCE and generalized SEQUENCE constraints which are based on network flows. Our propagator for the SEQUENCE constraint enforces domain consistency in $O(n^2)$ time down a branch of the search tree. This improves upon the best existing domain consistency algorithm by a factor of $O(\log n)$. We also introduced a soft version of the SEQUENCE constraint and propose an $O(n^2 \log n \log \log u)$ time domain consistency algorithm based on minimum cost network flows. These algorithms are derived from linear programs which represent a network flow. They differ from the flows used to propagate global constraints like GCC since the domains of the variables are encoded as costs on the edges rather than capacities. Such flows are efficient for maintaining bounds consistency over large domains. Experimental results demonstrate that the FB filtering algorithm is more robust than existing propagators. We conjecture that similar flow based propagators derived from linear programs may be useful for other global arithmetic constraints.

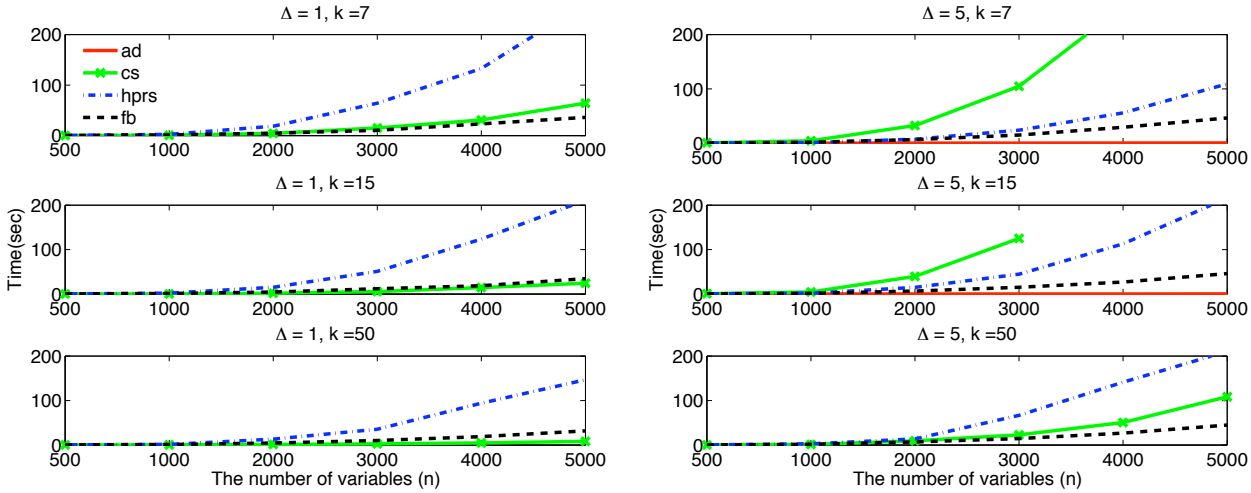


Figure 5. Randomly generated instances with a single SEQUENCE constraints for different combinations of Δ and k.

REFERENCES

[1] Jr. A.F. Veinott and H.M. Wagner, 'Optimal capacity scheduling I', *Operations Research*, **10**(4), 518–532, (1962).

[2] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin, *Network Flows: Theory, Algorithms, and Applications*, Prentice Hall, 1993.

[3] N. Beldiceanu and M. Carlsson, 'Revisiting the cardinality operator and introducing cardinality-path constraint family', in *Proc. of the Int. Conf. on Logic Programming*, pp. 59–73, (2001).

[4] Nicolas Beldiceanu, 'Global constraint catalog', T-2005-08, SICS Technical Report.

[5] Nicolas Beldiceanu and Evelyne Contejean, 'Introducing global constraints in CHIP', *Mathematical and Computer Modelling*, **12**, 97–123, (1994).

[6] Christian Bessiere, Emmanuel Hebrard, Brahim Hnich, Zeynep Kiziltan, and Toby Walsh, 'The slide meta-constraint', Technical report, (2007).

[7] R.E. Bixby and W.H. Cunningham, 'Converting linear programs to network problems', *Mathematics of Operations Research*, **5**, 321–357, (1980).

[8] K.S. Booth and G.S. Lueker, 'Testing for the consecutive ones property, interval graphs and graph planarity using PQ-tree algorithms', *Journal of Computer and Systems Sciences*, **13**, 335–379, (1976).

[9] S. Brand, N. Narodytska, C.-G. Quimper, P. Stuckey, and T. Walsh, 'Encodings of the sequence constraint', in *Proc. of the 13th Int. Conf. on Principles and Practices of Constraint Programming*, (2007).

[10] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, *Introduction to Algorithms, Second Edition*, The MIT Press, 2001.

[11] Scott Cotton and Oded Maler, 'Fast and flexible difference constraint propagation for DPLL(T)', in *Proc. of Theory and Applications of Satisfiability Testing (SAT-2006)*, pp. 170–183, (2006).

[12] Andrew V. Goldberg and Satish Rao, 'Beyond the flow decomposition barrier', *J. ACM*, **45**, 753–782, (1998).

[13] Willem-Jan van Hoeve, Gilles Pesant, Louis-Martin Rousseau, and Ashish Sabharwal, 'Revisiting the sequence constraint', in *Proc. of the 12th Int. Conf. on Principles and Practice of Constraint Programming*, pp. 620–634, (2006).

[14] Jean-Charles Régin, 'A filtering algorithm for constraints of difference in csps', in *Proc. of the 12th National Conf. on AI (AAAI'94)*, volume 1, pp. 362–367, (1994).

[15] Jean-Charles Régin, 'Generalized arc consistency for global cardinality constraint', in *Proc. of the 12th National Conf. on AI (AAAI'96)*, pp. 209–215, (1996).

[16] Jean-Charles Régin, 'Combination of among and cardinality constraints', in *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, Second International Conference*, volume 3524, pp. 288–303. Springer, (2005).

[17] Jean-Charles Régin and Jean-Francois Puget, 'A filtering algorithm for global sequencing constraints', in *Proc. of the 3th Int. Conf. on Principles and Practice of Constraint Programming*, pp. 32–46, (1997).

[18] Alexander Schrijver, *Theory of linear and integer programming*, John Wiley & Sons, Inc., 1986.

[19] Christine Solnon, Van Dat Cung, Alain Nguyen, and Christian Artigues, 'The car sequencing problem: overview of state-of-the-art methods and industrial case-study of the ROADEF'2005 challenge problem', *European Journal of Operational Research (EJOR)*, (January 2008). In press.

[20] Willem Jan van Hoeve, Gilles Pesant, and Louis-Martin Rousseau, 'On global warming: Flow-based soft global constraints', *J. Heuristics*, **12**(4-5), 347–373, (2006).

Modeling for Random Search with Distribution-Oriented Constraints

Anna Moss[1]

Abstract. Some application domains of Constraint Programming (CP) technology imply a fundamental randomness requirement regarding generated solutions. An example of such problem domain is random functional test generation where it is particularly important to provide good random distribution of solutions over the solution space in order to avoid validation holes. In such applications, solving is performed by means of random search in order to achieve certain distribution goals. In this context, the need arises to express random distribution requirements through the modeling means. In this paper, we address this need and present a generalized notion of constraints, where a constraint is defined not only by feasible tuples of variable assignments but also by a distribution over these tuples. The paper provides a formal definition of such distribution-oriented constraints and presents specific examples arising in the field of random functional test generation, namely, weighted enumeration constraint, random disjunction constraint and bias constraint. We show how the distribution-oriented constraints can be formulated in terms of traditional CP concepts using a combination of traditional constraints and search procedures. We further build a complete system of meta-constraints (negation, conjunction, disjunction, implication) defined over traditional and distribution-oriented constraints to allow modeling with this new type of constraints. To prove the robustness of our approach, we have implemented the meta-constraint system over distribution-oriented constraints using the ILOG CP tool.

1 INTRODUCTION

In some application areas of CP technology, there exists an inherent requirement for random sampling of solution space. Specifically, a solving algorithm is required to produce solutions that are randomly distributed over the solution space according to some predefined requirements on the distribution. Here, the distribution specifies the desired probability for each feasible solution to be generated by a randomized solving algorithm.

An example of a task requiring random sampling of solution space is random functional test generation. We proceed by describing this problem domain and the way it makes use of the CP technology. One of the major tasks in processor design cycle is design verification on the register transfer level (RTL). Methodologies commonly used to perform this task include simulation-based validation. In simulation-based validation, a large amount of functional tests is developed in an attempt to exercise various execution scenarios which could lead to bug detection. Clearly, due to size and complexity of modern processor designs, it is not possible to deterministically cover all possible test scenarios. Therefore, a commonly applied approach is to generate so called random directed tests. These tests are driven by constraints which allow a validation engineer to express test intention. On the other hand, these tests employ randomness in order to achieve good test space coverage with a reasonable amount of tests. The task of generating directed random tests is known as random functional test generation. In order to facilitate the work of validation engineers, automated test generation tools are being developed. In the recent years, the CP technology has been widely applied to random functional test generation ([1], [2], [3], [4]). Specifically, constraint modeling is used to enable declarative description of design under test as well as to allow a validation engineer to express test intention. CP search techniques are used to produce tests answering both architectural and test intention constraints.

From the description of random functional test generation task provided above it can be seen that achieving good random distribution of tests over the test space is one of the particularly important requirements in this task. Indeed, a failure to comply with this requirement would mean that some subspaces of the test space remain underrepresented leaving potential validation holes.

While complete random search algorithms received much attention lately (see [5] for a survey of research results in this area), the focus of this research has been on analyzing and improving the performance of complete search methods by introducing randomization rather than addressing the requirement for random sampling of solution space. Random sampling of solution space has been studied in the context of the SAT problem ([6]). The latter work proposed a method to achieve a near-uniform sampling by means of augmenting a given problem instance with random XOR constraints thus, in expectation, reducing the solution space to a single random solution. Some research has also been performed to address the problem of generating CSP solutions uniformly at random ([7]).

In this paper we address the requirement for random sampling of solution space from the aspect of constraint modeling. We argue that under this requirement the modeling language should be extended to allow random distribution notion to be expressed by means of constraints. This extension allows an agent building a CSP model (e.g., a validation engineer) to declaratively express desired distribution requirements while the details of underlying search algorithms remain transparent to the modeling agent. We introduce the notion of *distribution-oriented constraints*, which are defined both by tuples of feasible variable assignments, like the traditional constraints, and a random distribution over these tuples. We provide specific examples of such constraints, motivated by the needs of modeling for random functional test generation, namely, weighted enumeration constraint, random disjunction constraint

[1] Intel Corporation, Haifa, Israel, e-mail: anna.moss@intel.com

and bias constraint. Next, we show how this new type of constraints can be implemented in the framework of a traditional CP engine. Our representation of distribution-oriented constraints is comprised of traditional constraints and search procedures. To enable the use of distribution-oriented constraints in the traditional constraint modeling framework, we implement a complete meta-constraint system over these constraints, including negation, conjunction, disjunction and implication meta-constraints. To prove the feasibility of the proposed approach, we have implemented the modeling language extension with distribution-oriented constraints using ILOG CP engine ([8]).

The rest of the paper is organized as follows. Section 2 provides background definitions. In Section 3 we define distribution-oriented constraints and provide examples from random functional test generation domain. Section 4 describes our representation of distribution-oriented constraints in the traditional CP framework. It is followed by the description of complete meta-constraint system over distribution-oriented constraints in Section 5. We conclude in Section 6 with the summary of results and future work directions.

2 BACKGROUND

For the sake of completeness, in this section we provide the CP background required for the presentation of our results. An in-depth survey of this subject can be found in [9].

The CP paradigm comprises problem modeling as a CSP, constraint propagation, search algorithms, and heuristics. A CSP is defined by:

- a set of constrained variables. Each variable is associated with a (finite) domain defined as a collection of values that the variable is allowed to take;
- a set of constraints. A constraint is a relation defined on a subset of variables which restricts the combinations of values that the variables can take simultaneously.

A *solution* to a CSP is an assignment of values to variables so that each variable is assigned a value from its domain and all the constraints are satisfied.

The constraints referred to in the classical CSP definition above, are also known as *hard* constraints, in the sense that any feasible solution to the CSP must satisfy these constraints. A relaxation of this constraint type is a *soft* constraint which is allowed to be violated if it cannot be satisfied. CP with soft constraints is known as Partial Constraint Satisfaction ([10]). Different criteria have been proposed to measure the quality of solution for a CSP with soft constraints and many algorithms have been proposed for partial constraint satisfaction (see [11] and references therein for some examples). Another variation of CSP aimed at relaxing hard constraints is Fuzzy CSP (FCSP) ([12]). In FCSP, for each constraint, levels of preference between 0 and 1 are assigned to each variable assignment tuple. These levels indicate how "good" the assignment satisfies the constraint. A solution to FCSP is an assignment to all variables that maximizes the preference level for a constraint having the lowest satisfaction value.

A *meta-constraint* is a constraint that is composed of other constraints by using operations on constraints, e.g. conjunction, disjunction, negation or implication (for more details on this concept, see [8]).

A CSP formulation of a problem is processed by a constraint solver which attempts to find a solution using a search algorithm combined with reductions of variable domains based on constraint information. The latter mechanism is known as *constraint propagation*.

The *search space* is the Cartesian product of the variable domains. A complete search algorithm explores the search space systematically and is guaranteed to find a solution if such exists even though the time consumed by the search can sometimes be impractically large. At the first stage the search algorithm invokes initial constraint propagation to reduce variable domains defined in the CSP model. The algorithm then explores the search space by implicitly building a search tree. The nodes in such a tree correspond to so called *choice points* where search decisions can be made. The arcs descending from a choice point represent possible decisions. These decisions are taken according to some search strategy. For example, the possible search strategy is to pick a variable which has not been *fixed* yet (i.e. whose domain has not yet been reduced to a single value) and to try assigning each value in the domain in turn to the variable. In this case the search tree contains a choice point so that the arcs descending from this choice point correspond to possible value assignments to the variable. Each time a decision is taken at a choice point, domains of some variables get modified. These changes trigger the constraint propagation mechanism that ensures that each of the relevant constraints is propagated in respect to the new domains. A branch of the search tree *fails* if the domain of some variable becomes empty. In this case the search algorithm backtracks to one of the earlier choice points and tries to explore another branch. A solution is reached when all the variables get fixed.

Let Z be a search space. A *solution space* $S \subseteq Z$ is a set of all possible assignments to variables that are solutions to the CSP. A random distribution over solution space is a function $P: S \rightarrow [0..1]$ such that $\sum_{s \in S} P(s) = 1$. A randomized search algorithm is said to perform *random sampling of solution space* according to distribution P if the probability of the algorithm to generate the solution s is $P(s)$, for each $s \in S$. In practice, this means that if the search algorithm is executed multiple times on the same CSP, the fraction of runs that produce a specific solution s is $P(s)$, for each feasible solution s.

3 DISTRIBUTION-ORIENTED CONSTRAINTS

As argued in Section 1, in a task where modeling is performed for a problem to be solved by random search with an emphasis on the distribution of random solutions, a model should be able to specify not just the valid combinations of variable assignments, but also the desired distributions over these assignments. In other words, there is the need to extend the modeling capabilities to express the notion of random distribution over solution space. To address this need, we introduce the concept of a distribution-oriented constraint which extends the traditional constraint concept. In the sequel of this section we present a formal definition of distribution-oriented constraints followed by specific examples from random functional test generation domain.

3.1 Definition

A *distribution-oriented constraint* C over variables $X_1,...,X_n$ with domains $T_1,...,T_n$ respectively, is defined by

- a relation $R \subseteq T_1 \times T_2 \times ... \times T_n$; this relation defines assignment tuples to $X_1,...,X_n$ that satisfy the constraint C;
- a family D of a random distributions over R; a distribution $d \in D$ defines the probability of each n-tuple in R to occur as an assignment to $X_1,..., X_n$ in a random solution to a CSP involving the constraint C; the family D specifies all acceptable distributions.

We observe that the second part of the definition above allows to express any restriction on the desired distribution of tuples satisfying the constraint. For example, if one would like to imply that X_1 should be assigned a specific value $t \in T_1$ with probability 0.5, then the family D in the definition is the family of all distributions over R for which the total probability of tuples with X_1=t equals 0.5.

We further define that an assignment A to the variables $\boldsymbol{X_1,...,X_n}$ *satisfies* the distribution-oriented constraint C if and only if $A \in R$. Observe that according to this definition, the distribution part of the distribution-oriented constraint definition is similar to a soft constraint in the sense that there is no rigid requirement for valid solutions to satisfy the distribution restriction. The distribution requirement also bears some resemblance with preference levels for assignment tuples defined in FCSP. However, the meaning of a distribution is completely different from that of preference levels. In FCSP, preference levels indicate the quality of solution and serve to determine valid solutions, bearing no relation to random search. On the other hand, in the definition above, any tuple $A \in R$ satisfies the constraint equally well and can be part of a valid solution. Here, values associated with the tuples in a distribution $d \in D$ are meaningful only in the context of random search and indicate the desired probability for each tuple to appear in a random solution.

Note that a special case of the family D of random distributions is the family of all possible distributions over R. This implies no restriction at all on the distribution of variable assignments. Therefore, in this case we get the definition of a traditional constraint. In other words, the distribution-oriented constraint is an extension of the traditional constraint concept.

3.2 Examples

We proceed by giving specific examples of distribution-oriented constraints arising in the task of random functional test generation.

3.2.1 Weighted enumeration constraint

Suppose a validation engineer who builds constraints to express test intention wishes to imply that a mnemonic used in an instruction should be from the arithmetic instructions group {ADD, SUB, MULT, DIV} so that ADD will appear with probability of 0.4, SUB will appear with probability of 0.3, and each of MULT and DIV will be selected with probability of 0.15. We propose that in this case the validation engineer use the *weighted enumeration constraint* of the form:

WeightedEnumerationConstraint*(V,S,W)*

where V stands for a constraint variable, S denotes the set of possible values V is allowed to take and W is the distribution over S, indicating the weight, or probability, of each value in S.

In the example above, the instance of the weighted enumeration constraint is ***WeightedEnumerationConstraint***(mnemonicVar, {ADD, SUB, MULT, DIV}, {0.4, 0.3, 0.15, 0.15}).

Observe that the weighted enumeration constraint defined above is a distribution-oriented constraint, where the relation R is the membership constraint $V \in S$ and the family D of distributions contains the only distribution W.

3.2.2 Random disjunction constraint

In modeling for random functional test generation, when a disjunction constraint is applied, it is often desirable that each of the disjunction clauses will be represented in a random solution with some associated probability. Consider, for example, a test that targets specific memory regions. Suppose the memory regions are specified by the following constraint:

(addressVar in [0xA000,0xAFFF]) or (addressVar in [0xC1000,0xC3FFF]) or (addressVar in [0xFF000,0xFFFFF])

From the nature of the validation task, it is clear that for each of the three specified ranges, one would like to have a non-zero probability to get a random solution where addressVar assumes a value from that range (provided such solution is feasible with respect to other constraints). Unfortunately, this is not the case when a traditional disjunction constraint is used.

Following the argument above, we introduce the *random disjunction constraint* of the form

RandomDisjunctionConstraint(ConstraintList,P)

where *ConstraintList* denotes the list of constraints corresponding to disjunction clauses, and P is the list of probabilities corresponding to these clauses. A typical usage example of this constraint is when P implies equal probability for each of the disjunction clauses.

Note that random disjunction constraint is a distribution-oriented constraint. To see this, substitute the disjunction between constraints in *ConstraintList* for the relation R in the definition of distribution-oriented constraints. The family D of distributions contains all the distributions satisfying the probability requirements in P.

3.2.3 Bias constraint

Suppose one wishes to imply a distribution restriction over random solutions so that in a random solution a certain constraint C holds with probability p and its opposite holds with probability $(1-p)$. For example, one could imply that an instruction produced as a random solution to the corresponding CSP should have a memory operand with the probability of 50%. It would seem a straightforward approach to add the constraint C to a model with probability p and to add its opposite with probability $(1-p)$. However, in this case the constraint C or its opposite would be treated as hard constraints and could cause no solution result in the presence of contradicting constraints. However, the modeling intention here is to use the constraint C as a distribution bias and not as a hard constraint in the model.

Following this argument, we propose to use a distribution-oriented constraint we refer to as *bias constraint* of the form:

BiasConstraint(C, p)

where *C* is a constraint and *p* indicates the desired probability for *C* to hold in a random solution.

Though it is a little harder to see than in the previous two examples, the bias constraint is also a distribution-oriented constraint. In this case, the relation *R* corresponds to an always true constraint, that is, contains all possible tuples over domains of variables in *C*. The distribution family *D* contains all the distributions where the total probability of tuples satisfying *C* equals *p*.

4 CP-BASED IMPLEMENTATION OF DISTRIBUTION-ORIENTED CONSTRAINTS

The sequel of this paper is dedicated to implementation of modeling with distribution-oriented constraints within the traditional CP framework. In [4] we observe that there are two possible ways to approach the challenges posed by the specifics of CP for random functional test generation. The first way is to develop custom CP algorithms and tools to address the special features of the problem domain. The other way is to build upon the existing CP technology and extend a traditional CP engine to meet the new requirements for modeling and search. The same two approaches are applicable to modeling extension by distribution-oriented constraints. We follow the arguments in favor of the second approach presented in [4], and demonstrate how the traditional CP framework can be extended to support modeling with distribution-oriented constraints.

In the rest of this section we show how distribution-oriented constraints can be represented using traditional CP building blocks, namely, constraints and search procedures.

We introduce an extended constraint object capable of representing distribution-oriented constraints and results of operations on these constraints. The extended constraint is an entity $E(C,A,\hat{E})$ composed of a constraint *C*, a search procedure *A*, and an opposite extended constraint $\hat{E}$ (also represented in the same way). The opposite is required to enable the use of the distribution-oriented constraints in meta-constraints as explained in the next section, and it can be undefined for some extended constraints. When representing a traditional constraint *T*, the constraint component *C* is the constraint *T* itself; the search procedure component *A* is an empty procedure that does nothing; in case the opposite of T is defined, $\hat{E}$ is composed of the opposite constraint of T, an empty search procedure and *E*. When representing a distribution-oriented constraint, the constraint component is defined by the relation in the definition of distribution-oriented constraint; the search procedure *A* is responsible for adding relevant constraints to the search based on some (randomized) decisions; the opposite $\hat{E}$ can be defined or undefined in accordance with some agreed modeling conventions. For example, it may be agreed upon that an opposite of a distribution-oriented constraint is a traditional constraint that is opposite to the constraint part of distribution-oriented constraint, with no regard to its distribution part.

Extended constraints can be incorporated into the standard CP framework as follows. Constraint components *C* of extended constraints are added to the constraint model at the modeling stage. The search components *A* are integrated into the conventional search algorithm so that these search components are executed before the conventional search begins.

Next we will show how the distribution-oriented constraint examples described in Section 3 can be represented as an extended constraint $E(C,A,\hat{E})$. We start with the weighted enumeration constraint over a variable *V*, a value set *S* and weights *W* of elements in *S*. As explained above, the constraint component *C* of *E* is the membership constraint $V \in S$. The search procedure component *A* of *E* implements the following algorithm. The search algorithm draws an element from *S* at random according to probabilities specified in *W*, i.e. each element $s \in S$ can be chosen with probability *W(s)*. Then the search algorithm creates a choice point with two branches. The first branch adds the constraint *(V equals s)* to the search; the second branch removes *s* from *S*, distributes the weight of *s* proportionally between other elements and calls the same search procedure recursively with the reduced value set. If the reduced value set is empty, the procedure fails. We observe that the described representation guarantees that any random solution satisfies the membership constraint $V \in S$, that is the solution is valid according to our definition. As to the distribution requirement, it will be satisfied in case there are no constraints eliminating some values in *S*, otherwise the requested distribution will be approximated by redistributing the weights of forbidden elements between the remaining ones.

We proceed with the representation of the random disjunction constraint over the constraint list *ConstraintList* and list of probabilities *P*. The constraint component of the corresponding extended constraint is a disjunction between clauses in *ConstraintList*. The search procedure performs an algorithm, similar to that of the previous example. First, one of the disjunction clauses in ConstraintList is chosen at random so that each clause *C* is chosen with probability *P(C)*. Let *C'* be a selected clause. Then the search algorithm creates a choice point with two branches. The first branch adds the constraint *C'* to the search; the second branch removes *C'* from *ConstraintList*, distributes its probability proportionally between the remaining clauses and calls the same search procedure recursively with the reduced list of disjunction clauses. If the list of clauses is empty, the procedure fails. As in the previous case, any random solution following this representation satisfies the disjunction constraint, while the distribution can be influenced by other constraints in the model.

Finally, we describe how the bias constraint over constraint *C* and probability *p* can be represented by an extended constraint. The constraint component of *E* is an always true constraint, while the search procedure performs the following algorithm. First, the algorithm selects between *C* and its opposite so that *C* is selected with probability *p*. Then a choice point is created with two branches. The first branch adds the selected constraint to the search and the second branch adds its opposite to the search. Clearly, one of the branches will necessarily succeed so the bias constraint will not cause conflicts with other constraints. On the other hand, the described search procedure tries to guide the distribution of random solution according to the given requirement, while this guidance can be overruled by conflicting hard constraints.

5 COMPLETE META-CONSTRAINT SYSTEM OVER DISTRIBUTION-ORIENTED CONSTRAINTS

To enable modeling with distribution-oriented constraints the modeling framework must support meta-constraints built over these constraints. In this paper we address traditional operations on constraints supported in existing CP engines, namely negation, disjunction, conjunction and implication.

In this section we show how the extended constraint representation can support operations on constraints mentioned above. Specifically, we demonstrate that the result of each of these operations performed on extended constraints can also be represented as an extended constraint, thus providing a complete meta-constraint system.

5.1 Conjunction

We begin by representing the result of the conjunction operation on two extended constraints $E_1(C_1,A_1,\hat{E}_1)$ *and* $E_2(C_2,A_2,\hat{E}_2)$ as an extended constraint $E_c(C_c,A_c,\hat{E}_c(\hat{C}_c,\hat{A}_c,E_c))$. The resulting extended constraint is defined by $C_c = (C_1$ *and* $C_2)$, $A_c = (A_1$ *and* $A_2)$ and $\hat{E}_c = (\hat{E}_1$ *or* $\hat{E}_2)$. In the above, *and* operation on constraints is the standard constraint conjunction, $(A_1$ *and* $A_2)$ operation on search procedures results in a search procedure that first performs A_1 and then A_2 and the *or* operation on extended constraints results in an extended constraint as defined in Section 5.2.

5.2 Disjunction

Next, we show that the result of the disjunction operation on extended constraints $E_1(C_1,A_1,\hat{E}_1)$ *or* $E_2(C_2,A_2,\hat{E}_2)$ can also be represented as an extended constraint $E_d(C_d,A_d,\hat{E}_d(\hat{C}_d,\hat{A}_d,E_d))$. First, we define $C_d = (C_1$ *or* $C_2)$ and $\hat{E}_c = (\hat{E}_1$ *and* $\hat{E}_2)$. Here, *or* operation on constraints is the standard constraint disjunction, and the $(\hat{E}_1$ *and* $\hat{E}_2)$ operation on extended constraints results in an extended constraint as defined in Section 5.1. The definition of A_d, however, must be done carefully to avoid the situation when the constraint C_1 is accompanied by the search procedure A_2 or vice versa. We define the algorithm A_d as follows. The algorithm creates a choice point with two branches. At one branch, the constraint C_1 is added to the search and then the procedure A_1 is performed. At the other branch, the same action is taken with respect to C_2 and A_2. The order of the branches is determined at random.

5.3 Negation

We proceed with representing the negation of an extended constraint $E(C,A,\hat{E}(\hat{C},\hat{A},E))$ as an extended constraint $E_n(C_n,A_n,\hat{E}_n)$. In this case the definition is $E_n = \hat{E}(\hat{C},\hat{A},E)$ as the opposite extended constraint is already a part of representation of E.

5.4 Implication

Finally, we give the representation of the implication meta-constraint as an extended constraint. Since this operation can be formulated in terms of negation and disjunction, the representation of its result as an extended constraint follows from the previous subsections. Specifically, let $E_1(C_1,A_1,\hat{E}_1) \Rightarrow E_2(C_2,A_2,\hat{E}_2)$ be an implication defined over two extended constraints E_1 and E_2. The implication result is the extended constraint defined by $\hat{E}_1$ *or* E_2.

6 CONCLUSIONS

We have proposed an extension to the traditional modeling paradigm for constraint satisfaction problems where random sampling of the solution space is required, e.g. in the task of random functional test generation. Our extension allows to declaratively express random distribution requirements on solutions by means of model constraints. We formalized this approach by generalizing the traditional constraint concept and providing the definition of a distribution-oriented constraint. The paper presented the examples of such constraints arising in the domain of random functional test generation. We have shown how modeling with distribution-oriented constraints can be implemented within the traditional CP framework on top of an existing CP engine. Our implementation included extended constraint representation capable of representing distribution-oriented constraints and the support for complete meta-constraint system over this extended constraint representation. We demonstrated how the examples of distribution-oriented constraints given in this paper can be implemented within the proposed framework. To prove the robustness of our approach, we implemented this framework on top of ILOG CP engine.

The solving algorithms proposed in this paper in order to satisfy distribution-oriented constraints represent a naïve approach in the sense that no attempt is made to address the correlation between distribution requirements in different distribution-oriented constraints as well as the effects of hard constraints. Moreover, the quality of resulting distribution depends on the ordering of search procedures for different distribution-oriented constraints. A natural direction for the future work is to improve on the search algorithms. As a basis for this improvement, it is required to define quantitative measures of distribution quality that would enable comparison between different random search algorithms.

ACKNOWLEDGMENTS

The author would like to thank Boris Gutkovich and Asa Ben-Tzur for their valuable comments.

REFERENCES

[1] E. Bin, et al., 'Using a constraint satisfaction formulation and solution techniques for random test program generation', *IBM System Journal*, **41**(3), 386–402, (2002).

[2] Y. Naveh, et al., 'Constraint-based random stimuli generation for hardware verification', *AI Magazine*, **28**(3), 13–18, (2007).

[3] B. Gutkovich and A. Moss, 'CP with architectural state lookup for functional test generation', *11-th Annual IEEE International Workshop on High Level Design Validation and Test*, 111–118, (2006).

[4] A. Moss, 'Constraint patterns and search procedures for CP-based random test generation', In *Proceedings of the Third*

International Haifa Verification Conference (HVC 2007), *Lecture Notes in Computer Science*, **4899**, 86–103, (2008).

[5] C. Gomes, 'Randomized backtrack search', In Milano, M. (ed.): *Constraint and Integer Programming: Toward a Unified Methodology*, Kluwer, 233–283, (2003).

[6] C. Gomes, A. Sabharwal and B. Selman, 'Near-Uniform Sampling of Combinatorial Spaces Using XOR Constraints', In Schölkopf, B., et al. (ed.): *Advances in Neural Information Processing Systems 19*, MIT Press, 481–488, (2007).

[7] R. Dechter, et al., 'Generating random solutions for constraint satisfaction problems', In *Proceedings of AAAI*, (2002).

[8] *ILOG Solver 6.5 Reference Manual*, 2007.

[9] B.M. Smith, 'Modeling for constraint programming', *The 1-st Constraint Programming Summer School*, (2005).

[10] E. Freuder and R. Wallace, 'Partial constraint satisfaction', *Artificial Intelligence*, **58**, 21–70, (1992).

[11] T. Petit, C. Bessière and J.C. Régin, 'A general conflict-set based framework for partial constraint satisfaction', In *Proceedings of the 5-th International Workshop on Soft Constraints*, Kinsale, Ireland, (2003).

[12] D. Dubois, H. Fargier and H. Prade, 'The calculus of fuzzy restrictions as a basis for flexible constraint satisfaction', In *Proceedings of 2-nd IEEE International Conference on Fuzzy Systems*, **2**, 1131–1136, (1993).