2010 TRICS

3rd workshop on

Techniques
fo**R**
Implementing
Constraint programming
Systems

September 6, 2010

St. Andrews, Scotland

Workshop at the 16th International Conference on Principles and Practice of Constraint Programming

Proceedings

Preface

Aim

Constraint programming systems are software systems that support the modeling and solving of problems using constraint programming. Such systems include constraint programming libraries and runtime systems for constraint programming languages.
After two very successful workshops in 2000 (Singapore) and 2002 (Ithaca), this third iteration will again provide a forum for research in implementation of constraint programming systems.
The workshop encourages submissions and participation from all members of the CP community and all people interested in the implementation of constraint programming systems. It provides a place at CP where useful results, practical tricks and preliminary work can be presented, which by itself may not be sufficent for a submission to the full conference but is still of interest to other CP practitioners.

Topics

The topics of the workshop include

- data structures and algorithms for constraint solving
- parallelization approaches
- surveys and evaluation of often-used techniques
- machine learning approaches for constraint solver configuration
- debugging of constraint problems

Organization

The workshop is organized as a half-day workshop at the Sixteenth International Conference on Principles and Practice of Constraint Programming, CP 2010, and takes place in St. Andrews, Scotland, on September 6, 2010.

Workshop Organization

Organizers:

Christopher Jefferson, University of St. Andrews, UK
Peter Nightingale, University of St. Andrews, UK
Guido Tack, Katholieke Universiteit Leuven, Belgium

Program Committee:

Mats Carlsson, Swedish Institute of Computer Science, Sweden
Ian Gent, University of St. Andrews, UK
Youssef Hamadi, Microsoft Research, Cambridge, UK
Xavier Lorca, Ecole des Mines de Nantes, France
Christian Schulte, KTH - Royal Institute of Technology, Sweden
Peter Stuckey, University of Melbourne, Australia
Radoslaw Szymanek, École Polytechnique Fédérale de Lausanne, Switzerland
Pascal Van Hentenryck, Brown University, USA

Additional Reviewers:

Alejandro Arbelaez
Mikael Zayenz Lagerkvist

Workshop Program

Monday, September 6, 2010

09:00–09:10	**Opening remarks**
09:10–09:30	*Combining Parallel Search and Parallel Consistency in Constraint Programming* Carl Christian Rolf, Krzysztof Kuchcinski
09:30–09:50	*Distributed solving through model splitting* Lars Kotthoff, Neil C.A. Moore
09:50–10:10	*Experimental Evaluation of Branching Schemes for the CSP* Thanasis Balafoutis, Kostas Stergiou, Anastasia Paparrizou
10:10–10:30	*GenDebugger: An Explanation-based Constraint Debugger* *(System Demonstration)* Reuven Naveh, Giora Alexandron, Aaron Rich, Vitaly Lagoon
10:30–10:50	**Tea break**
10:50–11:10	*Implementing Efficient Propagation Control* Christian Schulte, Guido Tack
11:10–11:30	*Handling Heterogeneous Constraints in Revision Ordering Heuristics* Julien Vion, Sylvain Piechowiak
11:30–11:50	*Machine learning for constraint solver design – A case study for the alldifferent constraint* Ian Gent, Lars Kotthoff, Ian Miguel, Peter Nightingale
11:50	**Lightning talks session**
12:40	**Lunch**

Table of Contents

Experimental Evaluation of Branching Schemes for the CSP

Thanasis Balafoutis[1], Anastasia Paparrizou[2], and Kostas Stergiou[2]

[1] Department of Information and Communication Systems Engineering, University of the Aegean, Greece.
[2] Department of Informatics and Telecommunications Engineering, University of Western Macedonia, Greece.

Abstract. The search strategy of a CP solver is determined by the variable and value ordering heuristics it employs and by the branching scheme it follows. Although the effects of variable and value ordering heuristics on search effort have been widely studied, the effects of different branching schemes have received less attention. In this paper we study this effect through an experimental evaluation that includes standard branching schemes such as 2-way, *d*-way, and dichotomic domain splitting, as well as variations of set branching where branching is performed on sets of values. We also propose and evaluate a generic approach to set branching where the partition of a domain into sets is created using the scores assigned to values by a value ordering heuristic, and a clustering algorithm from machine learning. Experimental results demonstrate that although exponential differences between branching schemes, as predicted in theory between 2-way *d*-way branching, are not very common, still the choice of branching scheme can make quite a difference on certain classes of problems. Set branching methods are very competitive with 2-way branching and outperform it on some problem classes. A statistical analysis of the results reveals that our generic clustering-based set branching method is the best among the methods compared.

1 Introduction

Complete algorithms for CSPs are based on exhaustive backtracking search interleaved with constraint propagation. Search is typically guided by variable and value ordering heuristics and makes use of a specific branching scheme like 2-way or *d*-way branching. Although the impact of variable and value ordering heuristics on search performance are topics that have received very wide attention from the early days of CP, the impact of different branching schemes has not been as widely studied. As a result, the majority of modern finite domain CP solvers offer a wide range of variable and value ordering heuristics for the user/modeller to choose from, but at the same time they typically always employ 2-way branching. Some solvers allow for the user to implement different branching schemes, but it is not clear in which cases this may be desirable, and which particular scheme should be prefered.

In 2-way branching, after a variable x with domain $\{a_1, \dots, a_d\}$ is chosen, its values are assigned through a sequence of binary choices [14]. The first choice point creates two branches, corresponding to the assignment of a_1 to x (left branch) and the removal of a_1 from the domain of x (right branch). An alternative branching scheme which was extensively used in the past, and is still used by some solvers, is d-way branching. In this case, after variable x is selected, d branches are built, each one corresponding to one of the d possible value assignments of x. 2-way branching was described by Freuder and Sabin within the MAC algorithm [14] and in theory it can achieve exponential savings in search effort compared to d-way branching [8]. However, the few experimental studies comparing 2-way and d-way branching have not displayed significant differences between the two [11, 16]. Very recently Balafoutis and Stergiou showed that depending on the variable ordering heuristic used there can be from marginal to exponential differences between the two schemes [1].

Another technique that is sometimes used is dichotomic domain splitting [4]. This method originates from numerical CSPs and proceeds by splitting the current domain of the selected variable into two sets, usually based on the lexicographical ordering of the values. In this way branching is performed on the two created sets and the branching factor is reduced to two. Although domain splitting drastically reduces the branching factor, it can result in a much deeper search tree since the effects of propagation after a branching decision may be diminished.

In addition to these standard schemes, techniques that group together the values of the selected variable, and branch on these created groups instead of individual values, have been proposed [7, 10, 15, 2, 17, 9]. The criteria used for the grouping of values and the methods used to perform the grouping can be different, but all these techniques aim at reducing the size of the search tree. In this paper, following [9], we call any such method a *set branching* method.

Our first goal in this paper is to experimentally study the effect of different branching schemes for finite domain CSPs on search performance. Although some existing branching methods have been compared to one another (e.g. [16]), to our knowledge this is the first systematic evaluation of several existing alternatives. In addition, we propose and study a generic set branching method where the partition of a domain into sets is created using the scores assigned to values by a value ordering heuristic, and a clustering algorithm. Before employing such a method, two fundamental questions need to be adressed: What is the measure of similarity between values, and how do we partition domains using such a measure? Most of the approaches to set branching that have been proposed in the past have either used very strict measures of similarity or are problem specific. Our method offers a generic solution to both the problem of similarity evaluation and the partitioning of domains. For the former we exploit the information acquired from the value ordering heuristic, while for the latter we use a clustering algorithm from machine learning.

Experimental results from a wide range of benchmarks demonstrate that exponential differences between branching schemes, as predicted in theory between

2-way *d*-way, are not very common. But although the choice of branching scheme does not have as a profound effect as the choice of variable ordering heuristic, it can still make a difference. The generic set branching methods we evaluate outperform the standard 2-way branching scheme in many problem classes resulting in better average performance. It is notable that our clustering-based set branching method displays very promising results without any tuning of the clustering algorithm applied. Importantly, a statistical analysis of the experimental results reveals that this method is the best among the methods compared.

The rest of the paper is organized as follows. Section 2 gives necessary background. In Section 3 we discuss past work on set branching for CSPs and propose a new generic method for set branching. In Section 4 we report results from an experimental evaluation of the various branching schemes including a statistical analysis. Finally, in Section 5 we conclude.

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 $\{c_1, c_2, ..., c_e\}$ between variables in subsets of X. Each constraint $c \in C$ expresses a relation $rel(c)$ defining the variable assignment combinations that are allowed for the variables in the scope of the constraint *vars(c)*.

Complete search algorithms for CSPs are typically based on backtracking depth-first search where branching decisions (e.g. variable assignments) are interleaved with constraint propagation. Search is guided by variable ordering heuristics and value ordering heuristics.

One of the most efficient general purpose variable ordering heuristics that have been proposed is *dom/wdeg* [3]. This heuristic assigns a weight to each constraint, initially set to one. Each time a constraint causes a conflict, i.e. a domain wipeout, its weight is incremented by one. Each variable is associated with a *weighted degree*, which is the sum of the weights over all constraints involving the variable and at least another unassigned variable. The *dom/wdeg* heuristic chooses the variable with minimum ratio of current domain size to weighted degree.

A well-known generic value ordering heuristic for binary CSPs is Geelen's *promise* [6]. For each value $a \in D(x)$ this heuristic counts the number of values that are compatible with a in each future unassigned variable that x is constrained with. The product of these counts is the promise of a. The value with the maximum promise is selected.

3 Branching Schemes

From the early days of CSP research, search algorithms were usually implemented using either a *d-way* or a *2-way* branching scheme. The former works as follows. After a variable x with domain $D(x) = \{a_1, a_2, ..., a_d\}$ is selected, d branches are created, each one corresponding to a value assignment of x. In the first branch, value a_1 is assigned to x and constraint propagation is triggered. If this branch fails, a_1 is removed from $D(x)$. Then the assignment of a_2 to x is made (second branch), and so on. If all d branches fail then the algorithm backtracks. An example of a search tree explored with d-way branching is shown in Figure 1a.

In *2-way* branching, after a variable x and a value $a_i \in D(x)$ are selected, two branches are created. In the left branch a_i is assigned to x, or in other words the constraint $x{=}a_i$ is added to the problem and is propagated. In the right branch the constraint $x \neq a_i$ is added to the problem and is propagated. If there is no failure then any variable can be selected next (not necessarily x). If both branches fail then the algorithm backtracks. Figure 1b shows a search tree explored with 2-way branching.

There are two differences between these branching schemes. In 2-way branching, if the branch assigning a value a_i to a variable x fails then the removal of a_i from $D(x)$ is propagated. Instead, d-way branching tries the next available value a_j of $D(x)$. Note that the propagation of a_j subsumes the propagation of a_i's removal. In 2-way branching, after a failed branch corresponding to an assignment $x{=}a_i$, and assuming the removal of a_i from $D(x)$ is then propagated successfully, the algorithm can choose to branch on any variable (not necessarily x), according to the variable ordering heuristic. In d-way branching the algorithm has to again branch on x after $x{=}a_i$ fails.

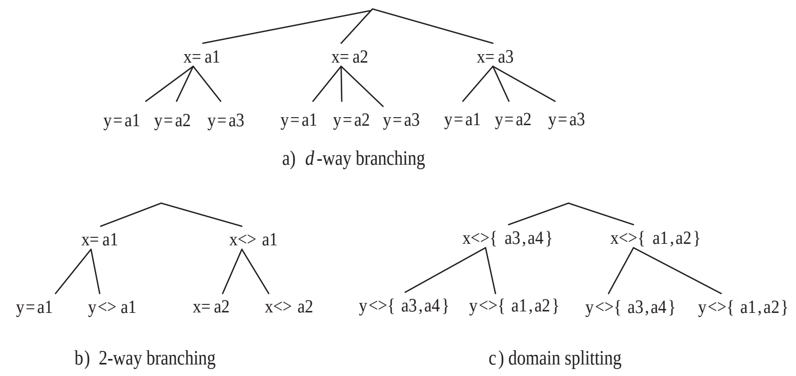

Fig. 1. Examples of search trees for the three branching schemes.

Another option, that originates from numerical CSPs, is dichotomic *domain splitting* [4]. This method proceeds by splitting the current domain of the selected variable into two sets, usually based on the lexicographical ordering of the values. Once the domain has been split, the second set of values is removed from the domain and this removal is propagated. In this way branching is performed on the two created sets and the branching factor is reduced to two. However, domain splitting tends to achieve weaker propagation compared to d-way and 2-way branching. So, although it drastically reduces the branching factor, it can result in a much deeper search tree. Domain splitting is mostly used on optimization problems and especially when the domains of the variables are very large. An example of a search tree explored with domain splitting is shown in Figure 1c.

3.1 Set Branching

Very recently, Kitching and Bacchus explored the applicability of *set branching* for constraint optimization problems [9]. The basic idea is to group together values that offer similar improvement to the currently computed bounds. In this way entire groups of values that offer no improvement to the bounds can be refuted, resulting in smaller tree sizes.

In this paper we use the term *set branching* to refer to any branching technique that, using some similarity criterion, identifies values that can be grouped together and branched on as a set. Dichotomic domain splitting and 2-way branching can be seen as manifestations of this generic method that use simple grouping criteria. Domain splitting creates two sets of values based on their lexicographical ordering. 2-way branching splits the domain into two sets where the first includes a single value and the second the rest of the values. In general, in order to define a set branching technique, two questions need to be addressed: *What is the measure of similarity between values, and how are domains partitioned using such a measure*?

The idea of set branching for CSPs has been explored in the past. Freuder introduced the notion of interchangeability, substitutability, and their weaker, but tractable, neighborhood versions as means to identify values with similar behavior [5]. Two values of a variable are neighborhhood interchangeable iff they have exactly the same supports in all constraints. One value a is neighborhood substitutable for another value b if the set of values inconsistent wth a is a subset of the values inconsistent with b. These notions were exploited, for example in [7, 2, 13], to group together values when branching and in this way perform set branching. The drawback of these techniques is that their conditions are too strong, as in many problems neighborhood interchangeable and substitutable values are very rare.

Larrosa investigated the merging of similar subproblems during search using forward checking [10]. According to this approach, values whose assignment leads to similar subproblems are grouped together and branched on as a set. Experiments performed on crossword puzzle generation problems displayed promising results. However, the measure of subproblem similarity and the algorithm used to partition the domains according to this measure are both problem specific.

Silaghi et al. proposed a method for partitioning the domains of variables based on the Cartesian product representation of the search space [15]. This method is particularly suitable for finding all solutions but it requires an explicit extensional representation of the constraints in the problem.

A generic and simple approach to set branching that can be applied on a wide range of problems was proposed by van Hoeve and Milano [17]. In this approach, values that are “tied” according to their value ordering heuristic score are grouped together and branching is performed on the sets of values created. Assignment of specific values to variables is postponed until lower levels of the search tree (which is also done in Larossa’s method). Experiments using both depth-first search and limited discrepancy search displayed promising results. However, this method relies heavily on the particular value ordering heuristic used and the number of ties produced by the value ordering heuristic, which can be quite low in many cases. Also, this method distinguishes between values that have very close but not equal scores and as a result such values will be placed into different sets. As noted in [17], the concept of a tie can be extended to refer to values having close scores. In this paper we explore this idea further.

3.2 Clustering for Set Branching

As we intend to apply set branching dynamically throughout search, after selecting a variable x with current domain $D(x) = \{a_1, \ldots, a_d\}$, we are faced with the following problem. We have to create a partition $S_{D(x)} = \{s_1, \ldots, s_m\}$ of $D(x)$ into m sets s.t. each value $a_i \in D(x)$ belongs to only one set $s_j \in S$. Ideally, we want all the values that have been assigned to a specific set to be similar according to some measure of similarity.

Following van Hoeve and Milano, we use a generic measure of similarity that is based on the score of the values according to a value ordering heuristic. In order to perform the dynamic partitioning of domains into sets, we propose the use of clustering algorithms from machine learning. Our approach can be summarized as follows. A value ordering heuristic is used to assign a score v_i to each value $a_i \in D(x)$. The collection of d items (values) and the matrix of their scores are given as input to a clustering algorithm. The output of the algorithm will be the partition $S_{D(x)} = \{s_1, \ldots, s_m\}$.

Compared to [17] our approach has the following potential benefits. First, not only will tied values be placed in the same set, but with high probability so will values that have very close scores. Hence, there will be fewer sets, resulting in lower branching factor. Second, in cases where there are no ties, the method of [17] uses d-way branching. In contrast, our approach will still partition the domain if there are groups of values with similar score.

The algorithm we currently use to create the clustering of values is *x-means* [12]. This is an extension of the well known k-means algorithm that is considerably faster and does not require to predetermine the desired number of clusters. The algorithm starts with randomly selected points (values in our case) as cluster centroids and iteratively improves the computed clustering until a fixpoint is reached. Several parameters of the algorithm can be tuned to give more accurate results on a specific application, including the starting centroids, the number of iterations, the measure of distance between points, etc. Although we intend to investigate this in the future, in the experiments reported below we use the Weka implementation of the x-means algorithm as is, without any tuning.

4 Experimental Evaluation

We have experimented with 350 instances from ten classes of real world, academic, patterned, and random CSPs taken from C.Lecoutre's XCSP repository. We included both satisfiable and unsatisfiable instances. Each selected instance involves constraints defined either in intension or in extension. The CSP solver used in our experiments is a generic solver and has been implemented in the Java programming language. This solver essentially implements the M(G)AC search algorithm, where (G)AC-3 is used for applying (G)AC. Since our solver does not yet support global constraints (apart from the table constraint) , we have left experiments with problems that include such constraints as future work. All experiments were run on an Intel dual core PC T4200 2GHz with 3GB RAM.

For a fair evaluation of the different branching schemes we use the same propagation method during search (arc consistency), the same variable ordering heuristic (*dom/wdeg* [3]) and value ordering heuristic (*Geelen's promise* [6]). The promise metric is calculated over all the visited nodes of the search tree. This penalizes run times and as a result may be inefficient in some problems, but for the purposes of this initial investigation we only wanted to use a reasonably sophisticated value ordering heuristic throughout all the tried instances. In the future we intend to experiment with different value ordering heuristics and study their effect on the performance of the clustering set branching method.

We compare the following branching schemes:

2-way Values are chosen in descending order of their promise.
***d*-way** Values are chosen in descending order of their promise.
domain splitting The values are ordered according to their promise and then the domain is split in half. The part with the top ranked values is tried first.
ties set branching This is the method of [17] where values with the same promise form a set. The sets are tried in descending order of promise.
clustering set branching This is our method where x-means is used to partition the domain into sets based on the promise of the values. The sets are tried in descending order of promise. Note that the clusters are linearly ordered since clustering is done over only one dimension.

The two set branching methods have been implemented using a 2-way and a *d*-way branching style, giving four alternatives. More specifically, past works on set branching for CSPs perform set branching using a *d*-way style. That is, once the partition of the domain $S_{D(x)} = \{s_1, \ldots, s_m\}$ is created, search proceeds by removing from $D(x)$ any value a, s.t. $a \notin s_1$, and propagating. If there is a failure, the same process is repeated for s_2 and so on. We have also implemented and evaluated 2-way style set branching. In this case the generated sets are tried in a series of binary choices. That is, after the reduction of $D(x)$ to s_1 fails, we propagate the removal from $D(x)$ of all the values in s_1. If this succeeds then we reduce $D(x)$ to s_2 and so on.

We must clarify here that in all the "2-way style" branching variants (domain splitting, ties, clustering) the set branching method allows to jump from one variable to another as standard 2-way branching does.

In addition, for domain splitting and the set branching methods we have tried two options: 1) Domain splitting (resp. set branching) is performed throughout search on all variables. 2) Domain splitting (resp. set branching) is performed on a variable only if its domain size is greater than a certain percentage of its original domain size. We have tried several values for this percentage, with 25% giving the best results. This can improve the performance of domain splitting by 30% on average, and it can offer (minor) improvement to set branching. Therefore, in the reported experiments with these methods Option 2 is followed.

Table 1. Cpu times (t), and nodes (n) from specific instances. Cpu times are in seconds. The best result for each instance is given in bold.

Problem Class		*d-way*	*2-way*	*dom split.*	*d-way ties set branch.*	*2-way ties set branch.*	*d-way clust. set branch.*	*2-way clust. set branch.*
*frb*35-17-2	t	**43.3**	98.4	954	60.1	98.3	134	154
(*sat*)	n	16241	45098	515909	27160	50713	58633	75743
*scen*3-*f*11	t	73.7	**6.9**	33.8	40.1	11.3	43.5	14.5
(*unsat*)	n	11056	1739	5318	11019	4021	13631	5705
pigeons-30-*ord*	t	2435	**572**	762	1259	773	1322	639
(*unsat*)	n	376384	135031	128286	338049	247792	364343	228190
*geo*50-20-*d*4-75-7	t	472	1338	2815	**190**	1309	365	543
(*sat*)	n	108027	404918	686333	58411	443724	111505	174716
langford-2-10	t	300	129	605	**108**	120	116	127
(*unsat*)	n	199104	247286	372733	199609	235912	203580	238314
driverw-09	t	177	145	243	**103**	164	180	143
(*sat*)	n	75625	93236	97180	46823	76510	77509	64798
qcp-15-120-6	t	23.8	12.4	26	28.8	**9.6**	133	94.6
(*sat*)	n	19074	20179	19353	33003	12019	136599	99847
qcp-15-120-8	t	50	35.4	53.2	44.4	130	**1.01**	**1.01**
(*sat*)	n	38227	49680	38551	46188	146342	845	845
*geo*50-20-*d*4-75-11	t	41.6	38.9	94.2	32.5	37.9	**12.2**	15.1
(*sat*)	n	9027	10044	21926	8990	12620	3486	5111
queensKnights-15-5-*add*	t	1506	1001	2245	1502	737	999	**594**
(*unsat*)	n	42154	15393	86199	42309	38836	28312	30890

Table 1 compares the various branching methods on specific instances from the tested problem classes. We display CPU times as well as nodes. A node in 2-way branching can correspond to a value assignment or to a value removal, while in d-way branching it can only correspond to a value assignment. Hence, they cannot be compared directly. The instances in this table are chosen to highlight the gaps in performance that can occur when using different branching schemes. As can be seen any method can be the best on a given instance, and there can be very considerable variance in the performance of the methods. For instance, clustering set branching can be very effective on certain problems (e.g. qcp-15-120-8) but it can also be quite ineffective on others (e.g. qcp-15-120-6).

However, these are some of the most 'extreme' instances. Exponential differences, as predicted between 2-way and *d*-way in theory, occured rarely[3].

Table 2. Average speed-up (positive values) or slow-down (negative values) achieved by 2-way branching compared to the other branching methods. Cpu time (t) in seconds and visited nodes (n) have been measured.

Problem Class	% *graph density*		*d-way*	*dom split.*	*d-way ties set branch.*	2-*way ties set branch.*	*d-way clust. set branch.*	2-*way clust. set branch.*
langford	1.045	t	2.88	5.08	-1.21	-1.11	-1.20	-1.04
(*unsat*)		n	-1.27	1.52	-1.26	-1.06	-1.23	-1.03
pigeons	1	t	1.13	1.24	-1.53	-1.89	-1.07	-1.32
(*unsat*)		n	-1.21	1.33	-1.7	-1.66	-1.25	-1.12
queensKnights	0.70	t	1.49	1.99	1.75	-1.21	-1.02	-1.48
(*unsat*)		n	2.85	4.96	3.47	3.04	1.87	2.39
forced random	0.65	t	-1.22	1.88	-1.30	-1.03	-1.14	1.14
(*sat*)		n	-1.41	1.52	-1.11	-1.1	-1.24	1.07
geometric	0.35	t	-2.48	2.07	-4.55	-1.03	-3.83	-2.58
(*sat*)		n	-3.02	1.79	-3.77	1.18	-3.53	-2.25
$qcp - qwh$	0.125	t	1.78	2.34	1.28	1.99	6.08	5.63
(*sat*)		n	-1.09	1.12	-1.06	1.5	4.08	3.84
driver	0.082	t	1.18	1.53	-1.33	1.10	1.21	1.00
(*sat*)		n	-1.23	-1.06	-1.71	-1.24	-1.23	-1.43
rlfap (*ScensMod*)	0.052	t	5.39	3.07	3.52	1.07	3.70	1.26
(*mixed*)		n	4.63	2.73	4.28	1.77	4.94	2.1
graphColoring	0.05	t	-1.50	1.01	-1.58	1.00	-1.49	-1.03
(*mixed*)		n	-1.28	1.15	-1.18	1.14	-1.17	-0.92

In Tables 2 and 3 we summarize the results of our experimental evaluation. We use 2-way branching as the standard all other branching methods are compared against. In Table 2 we give the average slow-down (or speed-up) of the methods compared to 2-way for each problem class (the two quasigroup classes qcp and qwh are grouped together). We have mostly selected problem classes that contain either only satisfiable or only unsatisfiable instances. However, we have also experimented with "mixed" problem classes. That is classes that contain both satisfiable and unsatisfiable instances. For example, on langford problems all instances are unsatisfiable and 2-way is 2.88 times better than *d*-way on average, while it is 1.2 times worse than *d*-way clustering set branching. As mentioned above, it is difficult to accurately compare the numbers of visited nodes under different branching schemes. However, in most problem classes the differences in Cpu times roughly reflect the differences in visited nodes.

In Table 3 we give the percentage of instances, over all the tried instances, where each method was faster (> 1), at least 2 times faster (> 2), and at least

[3] But this observation concerns the variable ordering heuristic and propagation method used here and may not generalize as shown in [1].

3 times faster (> 3) than 2-way branching. Similarly for instances where each method was slower by < 1, < 2, and < 3 times compared to 2-way.

Table 2 shows that although differences between methods can be quite large on single instances, the average differences between the most competitive methods are smaller. Dichotomic domain splitting is apparently the worst among the branching methods. However, it may fare better in problems with very large domain sizes[4]. Excluding domain splitting, the other methods are usually no more that 2 times better or worse than 2-way branching on average. But there are cases where even the average differences are quite large.

The set branching methods, and especially the *d*-way style ones, have slightly better or very close performance compared to 2-way branching on most classes. Also, these methods clearly outperform *d*-way branching. Interestingly, the set clustering methods are typically very competitive on the denser classes.

Table 3. % categorization of all tried instances according to the performance of the branching methods compared to 2-way branching.

Problem Class	*speedup*	*d-way*	*dom split*	*d-way ties set branch.*	*2-way ties set branch.*	*d-way clust. set branch.*	*2-way clust. set branch.*
all instances	>1	29%	11%	47%	68%	50%	45%
	>2	8%	0%	8%	2%	15%	16%
	>3	2%	0%	3%	0%	10%	6%
	<1	71%	89%	53%	32%	50%	55%
	<2	24%	56%	21%	2%	21%	15%
	<3	11%	34%	6%	3%	11%	6%

Table 3 shows that 2-way ties set branching is better than 2-way on most instances. However, the margins are usually small. This is because the number of ties that occur during search is usually low, meaning that 2-way ties set branching often emulates the standard 2-way scheme. The other set branching methods are better than 2-way on roughly half of the instances. However, they can be significantly better, and worse, on quite a few.

Table 4. Paired t-test measurements for evaluation of the significance of the experimental results. 2-way branching is compared with the other branching schemes.

	Mean	SD	t-value	95% C.I.
d-way	-29.8	341.7	-0.68	(-116, 57)
domain splitting	-241	456	-4.1	(-357, -125)
d-way ties set branching	9.48	326.3	0.23	(-73.3, 92.3)
2-way ties set branching	31.7	234	1.06	(-27.7, 91.1)
d-way clustering set branching	13.75	217.9	0.49	(-41.6, 69)
2-way clustering set branching	32.4	182.5	1.4	(-13.9, 78.7)

[4] Most domains included between 2 and 50 values, with maximum 225.

In order to obtain a global view and to evaluate the statistical significance of our experimental results, a set of paired t-tests were performed. In these tests we compared the CPU performance of the 2-way branching scheme against all the other branching schemes, over all the instances used in the experiments. We measured the mean difference, standard deviation, t-value and the 95% confidence interval. The risk level (called alpha level) was set to 0.05. Results are collected in Table 4.

As the results show, d-way branching and domain splitting are clearly inefficient compared to 2-way branching. The mean CPU reduction in the all set branching techniques is always greater than zero with 2-way clustering set branching being slightly better. However, the negative values at the confidence interval indicate that this reduction was not observed in all the tried instances. Although 2-way ties and clustering set branching achieve equivalent mean CPU reduction, the t-values score show that the spread (or variability) of the scores for 2-way clustering set branching is significantly higher compared to 2-way ties set branching. The t-value scores lead us to conclude that 2-way clustering set branching is a promising branching technique, since in our experiments it has displayed the best overall performance.

Finally, we have to mention that the number of clusters produced by x-means during search was usually quite low (2-3). In some cases, typically for small domain sizes, there was only one cluster generated because all values had similar score. In such a case our method switched to either d-way or 2-way branching depending on the style of set branching employed.

5 Conclusions

We performed an experimental evaluation of branching methods for CSPs including the commonly used 2-way and *d*-way schemes as well as other less widely used ones. We also proposed and evaluated a generic set branching method that partitions domains into sets of values by using information provided by the value ordering heuristic as input to a clustering algorithm. Results showed that set branching methods, including our approach, are competitive and often better compared to standard 2-way branching. We now plan to investigate ways to achieve more efficient domain partitions by automatically tuning the parameters of the clustering algorithm. Also, it would be interesting to study clustering of domains using information from multiple value ordering heuristics.

References

1. T. Balafoutis and K. Stergiou. Adaptive Branching for Constraint Satisfaction Problems. In *Proceedings of ECAI-2010*, 2010.
2. A. Beckwith and B. Choueiry. On the dynamic detection of interchangeability in finite constraint satisfaction problems. In *Proceedings of CP-01*, page 760, 2001.
3. F. Boussemart, F. Hemery, C. Lecoutre, and L. Sais. Boosting systematic search by weighting constraints. In *Proceedings of ECAI-04*, pages 146–150, 2004.

4. M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The Constraint Logic Programming Language CHIP. In *Proceedings of FGCS-88*, pages 693–702, 1988.
5. E. Freuder. Eliminating Interchangeable Values in Constraint Satisfaction Problems. In *Proceedings of AAAI-91*, pages 227–233, 1991.
6. P. A. Geelen. Dual viewpoint heuristics for binary constraint satisfaction problems. In *Proceedings of ECAI-92*, pages 31–35, 1992.
7. A. Haselbock. Exploiting interchangeabilities in constraint satisfaction problems. In *Proceedings of IJCAI-93*, pages 282–287, 1993.
8. J. Hwang and D. Mitchell. 2-Way vs. d-Way Branching for CSP. In *Proceedings of CP-2005*, pages 343–357, 2005.
9. M. Kitching and F. Bacchus. Set Branching in Constraint Optimization. In *Proceedings of IJCAI-09*, pages 532–537, 2009.
10. J. Larrosa. Merging constraint satisfaction problems to avoid redundant search. In *Proceedings of IJCAI-97*, pages 424–433, 1997.
11. V. Park. An empirical study of different branching strategies for constraint satisfaction problems, Master's thesis, University of London, 2004.
12. D. Pelleg and A. Moore. X-means: Extending K-means with Efficient Estimation of the Number of Clusters. In *Proceedings of ICML-2000*, pages 727–734, 2000.
13. S. Prestwich. Full Dynamic Interchangeability with Forward Checking and Arc Consistency. In *Proceedings of the ECAI Workshop on Modeling and Solving Problems With Constraints*, 2004.
14. D. Sabin and E.C. Freuder. Understanding and Improving the MAC Algorithm. In *Proceedings of CP-1997*, pages 167–181, 1997.
15. M. Silaghi, D. Sam-Haroud, and B. Faltings. Intelligent Domain Splitting for CSPs with Ordered Domains. In *Proceedings of CP-99*, pages 488–489, 1999.
16. B. Smith and P. Sturdy. Value Ordering for Finding All Solutions. In *Proceedings of IJCAI-05*, pages 311–316, 2005.
17. J. van Hoeve and M. Milano. Postponing Branching Decisions. In *Proceedings of ECAI-04*, pages 1105–1106, 2004.

Machine learning for constraint solver design

A case study for the `alldifferent` constraint

Ian Gent, Lars Kotthoff, Ian Miguel, and Peter Nightingale
{ipg,larsko,ianm,pn}@cs.st-andrews.ac.uk

University of St Andrews

Abstract. Constraint solvers are complex pieces of software which require many design decisions to be made by the implementer based on limited information. These decisions affect the performance of the finished solver significantly [16]. Once a design decision has been made, it cannot easily be reversed, although a different decision may be more appropriate for a particular problem.
We investigate using machine learning to make these decisions automatically depending on the problem to solve. We use the alldifferent constraint as a case study. Our system is capable of making non-trivial, multi-level decisions that improve over always making a default choice and can be implemented as part of a general-purpose constraint solver.

1 Introduction

Constraints are a natural, powerful means of representing and reasoning about combinatorial problems that impact all of our lives. Constraint solving is applied successfully in a wide variety of disciplines such as aviation, industrial design, banking, combinatorics and the chemical and steel industries, to name but a few examples.

A *constraint satisfaction problem* (CSP [3]) is a set of decision variables, each with an associated domain of potential values, and a set of constraints. An assignment maps a variable to a value from its domain. Each constraint specifies allowed combinations of assignments of values to a subset of the variables. A *solution* to a CSP is an assignment to all the variables that satisfies all the constraints. Solutions are typically found for CSPs through systematic search of possible assignments to variables. During search, constraint *propagation* algorithms are used. These propagators make inferences, usually recorded as domain reductions, based on the domains of the variables constrained and the assignments that satisfy the constraints. If at any point these inferences result in any variable having an empty domain then search backtracks and a new branch is considered.

When implementing constraint solvers and modelling constraint problems, many design decision have to be made – for example what level of consistency to enforce and what data structures to use to enable the solver to backtrack. These decisions have so far been made mostly manually. Making the "right" decision often depends on the experience of the person making it.

We approach this problem using machine learning. Given a particular problem class or problem instance, we want to decide *automatically* which design decisions to make. This improves over the current state of the art in two ways. First, we do not require humans to make a decision based on their experience and data available at that time. Second, we can change design decisions for particular problems.

Our system does not only improve the performance of constraint solving, but also makes it easier to apply constraint programming to domain-specific problems, especially for people with little or no experience in constraint programming. It represents a significant step towards Puget's "model and run" paradigm [23].

We demonstrate that we can approach machine learning as a "black box" and use generic techniques to increase the performance of the learned classifiers. The result is a system which is able to dynamically decide which implementation to use by looking at an unknown problem. The decision made is in general better than simply relying on a default choice and enables us to solve constraint problems faster.

2 Background

We are addressing an instance of the Algorithm Selection Problem [26], which, given variable performance among a set of algorithms, is to choose the best candidate for a particular problem instance. Machine learning is an established method of addressing this problem [17, 19]. Particularly relevant to our work are the machine learning approaches that have been taken to configure, to select among, and to tune the parameters of solvers in the related fields of mathematical programming, propositional satisfiability (SAT), and constraints.

Multi-tac [21] configures a constraint solver for a particular instance distribution. It makes informed choices about aspects of the solver such as the search heuristic and the level of constraint propagation. The Adaptive Constraint Engine [5] learns search heuristics from training instances. SATenstein [15] configures stochastic local search solvers for solving SAT problems.

An algorithm *portfolio* consists of a collection of algorithms, which can be selected and applied in parallel to an instance, or in some (possibly truncated) sequence. This approach has recently been used with great success in SATzilla [29] and CP Hydra [22]. In earlier work Borrett *et al* [2] employed a sequential portfolio of constraint solvers. Guerri and Milano [11] use a decision-tree based technique to select among a portfolio of constraint- and integer-programming based solution methods for the bid evaluation problem. Similarly, Gent *et al* [7] investigate decision trees to choose whether to use lazy constraint learning [9] or not.

Rather than select among a number of algorithms, it is also possible to learn parameter settings for a particular algorithm. Hutter *et al* [14] apply this method to local search. Ansotegui *et al* [1] employ a genetic algorithm to tune the parameters of both local and systematic SAT solvers.

The *alldifferent* constraint requires all variables which it is imposed on to be pairwise alldifferent. For example alldiff(x_1, x_2, x_3) enforces $x_1 \neq x_2$, $x_1 \neq x_3$ and $x_2 \neq x_3$.

There are many different ways to implement the alldifferent constraint. The naïve version decomposes the constraint and enforces disequality on each pair of variables. More sophisticated versions (e.g. [25]) consider the constraint as a whole and are able to do more propagation. For example an alldifferent constraint which involves four variables with the same three possible values each cannot be satisfied, but this knowledge cannot be derived when just considering the decomposition into pairs of variables. Further variants are discussed in [13].

Even when the high-level decision of how much propagation to do has been made, a low-level decision has to be made on how to implement the constraint. For an in-depth survey of the decisions involved, see [10].

We make both decisions and therefore combine the selection of an algorithm (the naïve implementation or the more sophisticated one) and the tuning of algorithm parameters (which one of the more sophisticated implementations to use). Note that we restrict the implementations to the ones that the Minion constraint solver [8] provides. In particular, it does not provide a bounds consistency propagator.

3 The benchmark instances and solvers

We evaluated the performance of the different versions of the alldifferent constraint on two different sets of problem instances. The first one was used for learning classifiers, the second one only for the evaluation of the learned classifiers.

The set we used for machine learning consisted of 277 benchmark instances from 14 different problem classes. It has been chosen to include as many instances as possible whatever our expectation of which version of the alldifferent constraint will perform best.

The set to evaluate the learned classifiers consisted of 1036 instances from 2 different problem classes that were not present in the set we used for machine learning. We chose this set for evaluation because the low number of different problem classes makes it unsuitable for training.

Our sources are Lecoutre's XCSP repository [18] and our own stock of CSP instances. The reference constraint solver used is Minion [8] version 0.9 and its default implementation of the alldifferent constraint `gacalldiff`. The experiments were run with binaries compiled with g++ version 4.4.3 and Boost version 1.40.0 on machines with 8 core Intel E5430 2.66GHz, 8GB RAM running CentOS with Linux kernel 2.6.18-164.6.1.el5 64Bit.

We imposed a time limit of 3600 seconds for each instance. The total number of instances that no solver could solve solve because of a time out was 66 for the first set and 26 for the second set. We took the median CPU time of 3 runs for each problem instance.

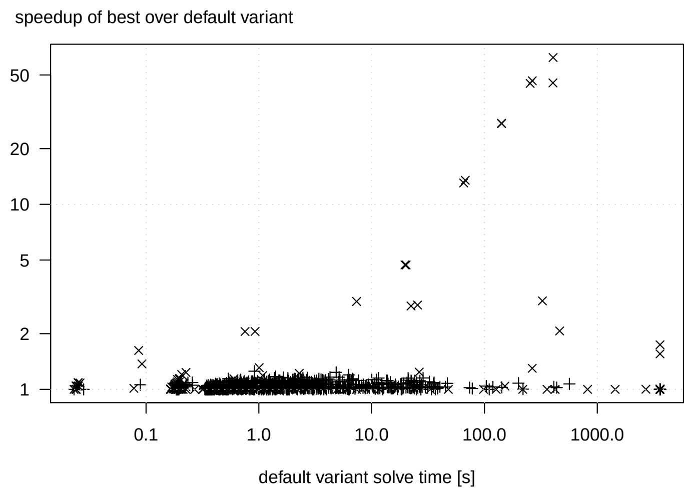

Fig. 1. Potential speedup a decision algorithm could achieve over always making the default decision. The crosses represent the instances of the first data set, the pluses the instances of the second data set. A speedup of one means that the default version of alldifferent is the fastest version, a speedup of two means that the fastest version of alldifferent is twice as fast as the default version.

As Figure 1 shows, adapting the implementation decision to the problem instead of always choosing a standard implementation has the potential of achieving significant speedups on some instances of the first set of benchmark instances and speedups of up to 1.2 on the second set.

We ran the problems with 9 different versions of the alldifferent constraint – the naïve version which is operationally equivalent to the binary decomposition and 8 different implementations of the more sophisticated version which achieves generalised arc consistency (see [10]). The amount of search done by the 8 versions which implement the more sophisticated algorithm was the same. The variables and values were searched in the order they were specified in in the model of the problem instance.

The instances, the binaries to run them, and everything else required to reproduce our results are available on request.

4 Instance attributes and their measurement

We measured 37 attributes of the problem instances. They describe a wide range of features such as constraint and variable statistics and a number of attributes based on the primal graph. The primal graph $g = \langle V, E \rangle$ has a vertex for every CSP variable, and two vertices are connected by an edge iff the two variables are in the scope of a constraint together.

Edge density The number of edges in g divided by the number of pairs of distinct vertices.

Clustering coefficient For a vertex v, the set of neighbours of v is $n(v)$. The edge density among the vertices $n(v)$ is calculated. The clustering coefficient is the mean average of this local edge density for all v [27] . It is intended to be a measure of the local cliqueness of the graph. This attribute has been used with machine learning for a model selection problem in constraint programming [11].

Normalised degree The normalised degree of a vertex is its degree divided by $|V|$. The minimum, maximum, mean and median normalised degree are used.

Normalised standard deviation of degree The standard deviation of vertex degree is normalised by dividing by $|V|$.

Width of ordering Each of our benchmark instances has an associated variable ordering. The width of a vertex v in an ordered graph is its number of *parents* (i.e. neighbours that precede v in the ordering). The width of the ordering is the maximum width over all vertices [3]. The width of the ordering normalised by the number of vertices was used.

Width of graph The width of a graph is the minimum width over all possible orderings. This can be calculated in polynomial time [3], and is related to some tractability results. The width of the graph normalised by the number of vertices was used.

Variable domains The quartiles and the mean value over the domains of all variables.

Constraint arity The quartiles and the mean of the arity of all constraints (the number of variables constrained by it), normalised by the number of constraints.

Multiple shared variables The proportion of pairs of constraints that share more than one variable.

Normalised mean constraints per variable For each variable, we count the number of constraints on the variable. The mean average is taken, and this is normalised by dividing by the number of constraints.

Ratio of auxiliary variables to other variables Auxiliary variables are introduced by decomposition of expressions in order to be able to express them in the language of the solver. We use the ratio of auxiliary variables to other variables.

Tightness The tightness of a constraint is the proportion of disallowed tuples. The tightness is estimated by sampling 1000 random tuples (that are valid w.r.t. variable domains) and testing if the tuple satisfies the constraint. The tightness quartiles and the mean tightness over all constraints is used.

Proportion of symmetric variables In many CSPs, the variables form equivalence classes where the number and type of constraints a variable is in are the same. For example in the CSP $x_1 \times x_2 = x_3, x_4 \times x_5 = x_6$, x_1, x_2, x_4, x_5 are all indistinguishable, as are x_3 and x_6. The first stage of the algorithm used by Nauty [20] detects this property. Given a partition of n variables generated by this algorithm, we transform this into a number between 0 and 1 by taking the proportion of all pairs of variables which are in the same part of the partition.

Alldifferent statistics The size of the union of all variable domains in an alldifferent constraint divided by the number of variables. This is a measure of how many assignments to all variables that satisfy the constraint there are. We used the quartiles and the mean over all alldifferent constraints.

In creating this set of attributes, we intended to cover a wide range of possible factors that affect the performance of different alldifferent implementations. Wherever possible, we normalised attributes that would be specific to problem instances of a particular size. This is based on the intuition that similar instances of different sizes are likely to behave similarly. Computing the features took 27 seconds per instance on average.

5 Learning a problem classifier

Before we used machine learning on the set of training instances, we annotated each problem instance with the alldifferent implementation that had the best performance on it according to the following criteria. If the naïve alldifferent implementation took less CPU time than all the other ones, it was chosen, else the implementation which had the best performance in terms of search nodes per second was chosen. All implementations except the naïve one explore the same search space. If no solver was able to solve the instance, we assigned a "don't know" annotation.

We used the WEKA [12] machine learning software through the R [24] interface to learn classifiers. We used almost all of the WEKA classifiers that were applicable to our problem – algorithms which generate decision rules, decision trees, Bayesian classifiers, nearest neighbour and neural networks. Our selection is broad and includes most major machine learning methodologies. The specific classifiers we used are `BayesNet`, `BFTree`, `ConjunctiveRule`, `DecisionTable`, `FT`, `HyperPipes`, `IBk`, `J48`, `J48graft`, `JRip`, `LADTree`, `MultilayerPerceptron`, `NBTree`, `OneR`, `PART`, `RandomForest`, `RandomTree`, `REPTree` and `ZeroR`, all of which are described in [28].

For all of these algorithms, we used the default parameters provided by WEKA. While the performance would have been improved by carefully tuning those parameters, a lot of effort and knowledge is required to do so. Instead, we used the standard parameter configuration which is applicable for other machine learning problems as well and not specific to this paper.

The problem of classifying problem instances here is different to normal machine learning classification problems. We do not particularly care about classifying as many instances as possible correctly; we rather care that the instances that are important to us are classified correctly. The higher the potential gain is for an instance, the more important it is to us. If, for example, the difference between making the right and the wrong decision means a difference in CPU time of 1%, we do not care whether the instance is classified correctly or not. If the difference is several orders of magnitude on the other hand, we really do want this instance to be classified correctly.

Based on this observation, we decided to measure the performance of the learned classifiers not in terms of the usual machine learning performance measures, but in terms of misclassification penalty [29]. The misclassification penalty is the additional CPU time we require to solve a problem instance when choosing to solve it with a solver that is not the fastest one. If the selected solver was not able to solve the problem, we assumed the timeout of 3600 seconds minus the CPU time the fastest solver took to be the misclassification penalty. This only gives the lower bound, but the correct value cannot be estimated easily.

We furthermore decided to assign the maximum misclassification penalty (or the maximum possible gain), cf. Figure 1 as a cost to each instance as follows. To bias the WEKA classifiers towards the instances we care about most, we used the common technique of duplicating instances [28]. Each instance appeared in the new data set $1 + \lceil \log_2(\texttt{cost}) \rceil$ times. The particular formula to determine how often each instance occurs was chosen empirically such that instances with a low cost are not disregarded completely, but instances with a high cost are much more important. Each instance will be in the data set used for training the machine learning classifiers at least once and at most 13 times for a theoretic maximum cost of 3600.

To achieve multi-level classification, each individual classifier below consists of a combination of classifiers. First we make the decision whether to use the alldifferent version equivalent to the binary decomposition or the other one, then, based on the previous decision, we decide which specific version of the alldifferent constraint to use.

Table 1 shows the total misclassification penalty for all classifiers with and without instance duplication on the first data set. It clearly shows that our cost model improves the performance significantly in terms of misclassification penalty for almost all classifiers.

	misclass. penalty [s]			misclass. penalty [s]	
classifier	all equal	cost model	classifier	all equal	cost model
BayesNet	1494	3.9	LADTree	8.4	6.5
BFTree	8.4	1.1	MultilayerPerceptron	249	8.5
ConjunctiveRule	2300	1433	NBTree	9	1.3
DecisionTable	249	1.6	OneR	69.5	409
FT	248	1.2	PART	5.9	1
HyperPipes	867	867	RandomForest	41.9	0.9
IBk	109	109	RandomTree	1	1
J48	8.2	1.2	REPTree	1099	10.8
J48graft	8.2	1.2	ZeroR	2304	2304
JRip	283	1.3			

Table 1. Misclassification penalty for all classifiers with and without instances duplicated according to their cost in the training data set. All numbers are rounded.

For each classifier, we did stratified n-fold cross-validation – the original data set is split into n parts of roughly equal size. Each of the n partitions is in turn used for testing. The remaining $n - 1$ partitions are used for training. In the end, every instance will have been used for both training and testing in different runs [28]. Stratified cross-validation ensures that the ratio of the different classification categories in each subset is roughly equal to the ratio in the whole set. If, for example, about 50% of all problem instances in the whole data are solved fastest with the naïve implementation, it will be about 50% of the instances in each subset as well.

There are several problems we faced when generating the classifiers. First, we do not know which one of the machine learning algorithms was suited best for our classification problem; indeed we do not know whether the features of the problem instances we measured are able to capture the factors which affect the performance of each individual implementation at all. Second, the learned classifiers could be overfitted. We could evaluate the performance of each classifier on the second set of problem instances and compare it to the performance during machine learning to assess whether it might be overfitted. Even if we were able to reliably detect overfitting this way, it is not obvious how we would change or retrain the classifier to remove the overfitting. Instead, we decided to use all classifiers – for each machine learning algorithm the n different classifiers created during the n-fold cross-validation and the classifiers created by each different machine learning algorithm.

We decided to use three-fold cross-validation as an acceptable compromise between trying to avoid overfitting and time required to compute and run the classifiers. We combine the decisions of the individual classifiers by majority vote. The technique of combining the decisions of several classifiers was introduced in [6] and formalised in [4].

Table 2 shows the overall performance of our meta-classifier compared to the best and worst individual classifier for each set and several other hypothetical classifiers. Our meta-classifier outperforms a classifier which always makes the default decision even on the second set of problem instances. This set is an extreme case because just making the default choice is almost always the best choice – the misclassification penalty for the default choice classifier is extremely low given the large number of instances. Even though there is only very little room for improvement (cf. Figure 1), we achieve some of it.

It also shows that the classifiers we have learned on a data set that contains problem instances from many problem classes can be applied to a different data set with instances from different problem classes and still achieve a performance improvement. Based on this observation, we suggest that our meta-classifier is generally applicable.

Another observation we made is that the performance of the meta-classifier does not suffer even if a large number of the classifiers that it combines perform badly individually. This suggests that the classifiers complement each other – the set of instances that each one misclassifies are different for each classifier. Note also that the classifier which performs best on one set of instances is not

	misclassification penalty [s]			
	instance set 1		instance set 2	
classifier	all features	cheap features	all features	cheap features
oracle	0	0	0	0
anti-oracle	19993	19993	47144	47144
default decision	**2304**	**2304**	**223**	**223**
random decision	5550	5550	564	564
best classifier on set 1	0.998	0.994	131	220.3
worst classifier on set 1	2304	2304	223	223
best classifier on set 2	0.998	61.66	131	186
worst classifier on set 2	1.34	1.44	621	610
meta-classifier	**1.16**	**0.996**	**220**	**222.95**

Table 2. Summary of classifier performance on both sets of benchmarks in terms of total misclassification penalty in seconds. We first evaluated the performance using the full set of features described in Section 4, then using only the cheap features. The oracle classifier always makes the right decision, the anti-oracle always the worst possible wrong decision. The "default decision" classifier always makes the same decision and the "random decision" one chooses one of the possibilities at random. Three-fold cross-validation was used. All numbers are rounded.

necessarily the best performer on the other set of instances. The same observation can be made for the classifier with the worst performance on one of the instance sets. This means that we cannot simply choose "the best" classifier or discard "the worst" for a given set of training instances. Table 3 provides further evidence for this. The individual best and worst classifiers vary not only with the data set, but also with the set of features used.

	instance set 1		instance set 2	
	all features	cheap features	all features	cheap features
best classifier	`IBk`	`BFTree`	`IBk`	`BayesNet`
worst classifier	`ZeroR`	`ZeroR`	`LADTree`	`LADTree`

Table 3. Individual best and worst classifiers for the different data and feature sets for the numbers presented in Table 2.

The time required to compute the features was 27 seconds per instance on average, and it took 0.2 seconds per instance on average to run the classifiers and combine their decisions. If we take this time into account, our system is slower than just using the default implementation. This is mostly because of the cost of computing all the features required to make the decision. We do however learn good classifiers in the sense that the decision they make is better than just using the standard implementation.

We now focus on making a decision as quickly as possible. Most of the time required to make the decision is spent computing the features that the classifiers need. We removed the most expensive features – all the properties of the primal graph described in Section 4 apart from edge density.

The results for the reduced set of features are shown in Table 2 as well. The performance is not significantly worse and even better on the first set of instances, but the time required to compute all the features is only about 3 seconds per instance. On the first set of benchmarks, we solve each instance on average 8 seconds faster using our system (misclassification penalty of default decision minus that of our system divided by the number of instances in the set). We are therefore left with a performance improvement of an average of 5 seconds per instance. On the second set, we cannot reasonably expect a performance improvement – the perfect oracle classifier only achieves about 0.2 seconds per instance on average.

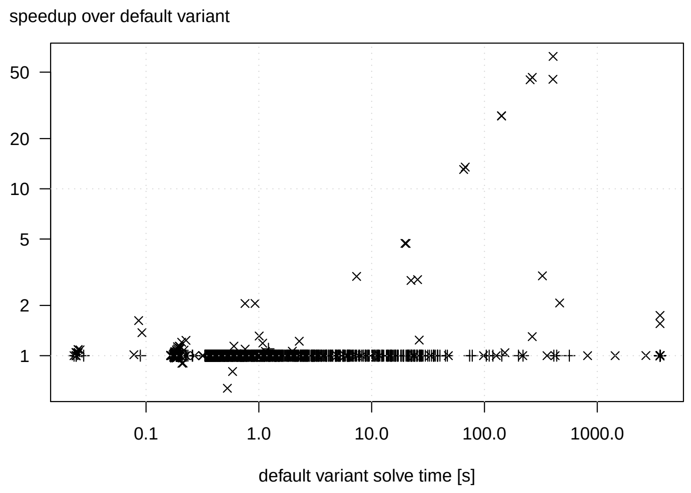

Fig. 2. Speedup achieved by the meta-classifier using the set of cheaply-computable features. The figure does not take the overhead of computing the features and running the classifiers into account. The crosses represent the instances of the first data set, the pluses the instances of the second data set.

Figure 2 revisits Figure 1 and shows the actual speedup our meta-classifier achieves for each instance. It convincingly illustrates the quality of our classifier. The instances where we suffer a slowdown are ones that are solved almost instantaneously, whereas the correctly classified instances are the hard ones that we care about most. In particular the instances where a large speedup can be gained are classified correctly by our system.

6 Conclusions and future work

We have applied machine learning to a complex decision problem in constraint programming. To facilitate this, we evaluated the performance of constraint solvers representing all the decisions on two large sets of problem instances. We have demonstrated that training a set of classifiers without intrinsic knowledge about each individual one and combining their decisions can improve performance significantly over always making a default decision. In particular, our combined classifier is almost as good as the best classifier in the set and much better than the worst classifier while mitigating the need to select and tune an individual classifier.

We have conclusively shown that we can improve significantly on default decisions suggested in the state-of-the-art literature using a relatively simple and generic procedure. We provide strong evidence for the general applicability of a set of classifiers learned on a training set to sets of new, unknown instances. We identified several problems with using machine learning to make constraint programming decisions and successfully solved them.

Our system achieves performance improvements even taking the time it takes to compute the features and run the learned classifiers into account. For atypical sets of benchmarks, where always making the default decision is the right choice in almost all of the cases, we are not able to compensate for this overhead, but we are confident that we can achieve a real speedup on average.

We have identified two major directions for future research. First, it would be beneficial to analyse the individual machine learning algorithms and evaluate their suitability for our decision problem. This would enable us to make a more informed decision about which ones to use for our purposes and may suggest opportunities for improving them.

Second, selecting which features of problem instances to compute is a non-trivial choice because of the different cost and benefit associated with each one. The classifiers we learned on the reduced set of features did not seem to suffer significantly in terms of performance. Being able to assess the benefit of each individual feature towards a classifier and contrast that to the cost of computing it would enable us to make decisions of equal quality cheaper.

Acknowledgements

The authors thank Chris Jefferson for providing some of the feature descriptions. We thank Jesse Hoey for useful discussions about machine learning and the anonymous reviewers for their feedback. Peter Nightingale is supported by EPSRC grants EP/H004092/1 and EP/E030394/1. Lars Kotthoff is supported by a SICSA studentship.

References

1. Ansótegui, C., Sellmann, M., Tierney, K.: A gender-based genetic algorithm for the automatic configuration of algorithms. In: CP. pp. 142–157 (2009)

2. Borrett, J., Tsang, E., Walsh, N.: Adaptive constraint satisfaction: The quickest first principle. In: ECAI. pp. 160–164 (1996)
3. Dechter, R.: Constraint Processing. Elsevier Science (2003)
4. Dietterich, T.G.: Ensemble methods in machine learning. In: First International Workshop on Multiple Classifier Systems. pp. 1–15 (2000)
5. Epstein, S., Freuder, E., Wallace, R., Morozov, A., Samuels, B.: The adaptive constraint engine. In: CP. pp. 525–542 (2002)
6. Freund, Y., Schapire, R.E.: A decision-theoretic generalization of on-line learning and an application to boosting. In: EuroCOLT. pp. 23–37 (1995)
7. Gent, I., Jefferson, C., Kotthoff, L., Miguel, I., Moore, N., Nightingale, P., Petrie, K.: Learning when to use lazy learning in constraint solving. In: ECAI (2010)
8. Gent, I., Jefferson, C., Miguel, I.: Minion: A fast scalable constraint solver. In: ECAI. pp. 98–102 (2006)
9. Gent, I., Miguel, I., Moore, N.: Lazy explanations for constraint propagator. In: PADL (2010)
10. Gent, I., Miguel, I., Nightingale, P.: Generalised arc consistency for the alldifferent constraint: An empirical survey. Artif. Intell. 172(18), 1973–2000 (2008)
11. Guerri, A., Milano, M.: Learning techniques for automatic algorithm portfolio selection. In: ECAI. pp. 475–479 (2004)
12. Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., Witten, I.: The WEKA data mining software: An update. SIGKDD Explorations 11(1) (2009)
13. van Hoeve, W.J.: The alldifferent Constraint: A Survey (2001)
14. Hutter, F., Hamadi, Y., Hoos, H., Leyton-Brown, K.: Performance prediction and automated tuning of randomized and parametric algorithms. In: CP. pp. 213–228 (2006)
15. KhudaBukhsh, A., Xu, L., Hoos, H., Leyton-Brown, K.: SATenstein: Automatically building local search SAT solvers from components. In: IJCAI. pp. 517–524 (2009)
16. Kotthoff, L.: Constraint solvers: An empirical evaluation of design decisions. CIRCA preprint (2009), http://www-circa.mcs.st-and.ac.uk/Preprints/solver-design.pdf
17. Lagoudakis, M., Littman, M.: Reinforcement learning for algorithm selection. In: AAAI/IAAI. p. 1081 (2000)
18. Lecoutre, C.: XCSP benchmarks. http://tinyurl.com/y6hpphs (June 2010)
19. Leyton-Brown, K., Nudelman, E., Andrew, G., McFadden, J., Shoham, Y.: A portfolio approach to algorithm selection. In: IJCAI. pp. 1542–1543 (2003)
20. McKay, B.: Practical graph isomorphism. In: Numerical mathematics and computing, Proc. 10th Manitoba Conf., Winnipeg/Manitoba 1980, Congr. Numerantium 30. pp. 45–87 (1981), see also http://cs.anu.edu.au/people/bdm/nauty
21. Minton, S.: Automatically configuring constraint satisfaction programs: A case study. Constraints 1(1/2), 7–43 (1996)
22. O'Mahony, E., Hebrard, E., Holland, A., Nugent, C., O'Sullivan, B.: Using case-based reasoning in an algorithm portfolio for constraint solving. In: 19th Irish Conference on AI (2008)
23. Puget, J.F.: Constraint programming next challenge: Simplicity of use. In: CP. pp. 5–8 (2004)
24. R Development Core Team: R: A Language and Environment for Statistical Computing. R Foundation for Statistical Computing (2009)
25. Régin, J.C.: A filtering algorithm for constraints of difference in CSPs. In: AAAI. pp. 362–367 (1994)
26. Rice, J.: The algorithm selection problem. Advances in Computers 15, 65–118 (1976)

27. Watts, D., Strogatz, S.: Collective dynamics of 'small-world' networks. Nature 393, 440–442 (1998)
28. Witten, I., Frank, E.: Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations. Morgan Kaufmann (2005)
29. Xu, L., Hutter, F., Hoos, H., Leyton-Brown, K.: SATzilla: Portfolio-based algorithm selection for SAT. J. Artif. Intell. Res. (JAIR) 32, 565–606 (2008)

Distributed solving through model splitting

Lars Kotthoff and Neil C.A. Moore
{larsko,ncam}@cs.st-andrews.ac.uk

University of St Andrews

Abstract. Constraint problems can be trivially solved in parallel by exploring different branches of the search tree concurrently. Previous approaches have focused on implementing this functionality in the solver, more or less transparently to the user. We propose a new approach, which modifies the constraint model of the problem. An existing model is split into new models with added constraints that partition the search space. Optionally, additional constraints are imposed that rule out the search already done. The advantages of our approach are that it can be implemented easily, computations can be stopped and restarted, moved to different machines and indeed solved on machines which are not able to communicate with each other at all.

1 Introduction

Constraint problems are typically solved by searching through the possible assignments of values to variables. After each such assignment, propagation can rule out possible future assignments based on past assignments and the constraints. This process builds a search tree that explores the space of possible (partial) solutions to the constraint problem.

There are two different ways to build up these search trees – n-way branching and 2-way branching. This refers to the number of new branches which are explored after each node. In n-way branching, all the n possible assignments to the next variable are branched on. In 2-way branching, there are two branches. The left branch is of the form $x = y$ where x is a variable and y is a value from its domain. The right branch is of the form $x \neq y$.

The more commonly used way is 2-way branching, implemented for example in the Minion constraint solver [5][1]. However, regardless of the way the branching is done, exploring the branches can be done concurrently. No information between the branches needs to be exchanged in order to find a solution to the problem.

We exploit this fact by, given the model of a constraint problem, generating new models which partition the remaining search space. These models can then be solved independently. We furthermore represent the state of the search by adding additional constraints such that the splitting of the model can occur at any point during search. The new models can be resumed, taking advantage of both the splitting of the search space and the search already performed.

[1] http://minion.sf.net

2 Background

The parallelisation of depth-first search has been the subject of much research in the past. The first papers on the subject study the distribution over various specific hardware architectures and investigate how to achieve good load balancing [13, 7]. Distributed solving of constraint problems specifically was first explored only a few years later [2].

Backtracking search in a distributed setting has also been investigated by several authors [12, 15]. A special variant for distributed scenarios, asynchronous backtracking, was proposed in [17]. Yokoo *et al* formalise the distributed constraint satisfaction problem and present algorithms for solving it [18].

Schulte presents the architecture of a system that uses networked computers [16]. The focus of his approach is to provide a high-level and reusable design for parallel search and achieve a good speedup compared to sequential solving rather than good resource utilisation. More recent papers have explored how to transparently parallelise search without having to modify existing code [10].

Most of the existing work is concerned with the problem of effectively distributing the workload such that every compute node is kept busy. The most prevalent technique used to achieve this is work stealing. The compute nodes communicate with each other and nodes which are idle request a part of the work that a busy node is doing. Blumofe and Leiserson propose and discuss a work stealing scheduler for multithreaded computations in [1]. Rolf and Kuchcinski investigate different algorithms for load balancing and work stealing in the specific context of distributed constraint solving [14].

Several frameworks for distributed constraint solving have been proposed and implemented, e.g. FRODO [11], DisChoco [3] and Disolver [6]. All of these approaches have in common that the systems to solve constraint problems are modified or augmented to support distribution of parts of the problem across and communication between multiple compute nodes. The constraint model of the problem remains unchanged however; no special constructs have to be used to take advantage of distributed solving. All parallelisation is handled in the respective solver. This does not preclude the use of an entirely different model of the problem to be solved for the distributed case in order to improve efficiency, but in general these solvers are able to solve the same model both with a single executor and distributed across several executors.

The decomposition of constraint problems into subproblems which can be solved independently has been proposed in [9], albeit in a different context. In this work, we explore the use of this technique for parallelisation. A similar approach was taken in [14], but requires parallelisation support in the solver.

3 Model splitting

We now describe our new approach to the distributed solving of constraint problems which modifies the constraint solver to modify the constraint model and does not require explicit parallelisation support in the solver.

Before splitting, the solver is stopped. As well as stopping, it is designed to output *restart nogoods* for the problem in the solver's own input language [8]. These constraints, when added to the problem, will prevent the search space just explored from being repeated in any split model[2].

To split the search space for an existing model, partition the domain for the variable currently under consideration into n pieces of roughly equal size. Then create n new models and to each in turn add constraints ruling out $n-1$ partitions of that domain. Each one of these models restricts the possible assignments to the current variable to one nth of its domain.

As an example, consider the case $n = 2$. If the variable under consideration is x and its domain is $\{1, 2, 3, 4\}$, we generate 2 new models. One of them has the constraint $x \leq 2$ added and the other one $x \geq 3$. Thus, solving the first model will try the values 1 and 2 for x, whereas the second model will try 3 and 4.

The main problem when splitting constraint problems into parts that can be solved in parallel is that the size of the search space for each of the splits is impossible to predict reliably. This directly affects the effectiveness of the splitting however – if the search space is distributed unevenly, some of the workers will be idle while the others do most of the work.

We address this problem by providing the ability to split a constraint model after search has started. The approach is very similar to the one explained above. The only difference is that in addition to the constraints that partition the search space, we also add constraints that rule out the search space that has been explored already.

Assume for example that we are doing 2-way branching, the variable currently under consideration is again x with domain $\{1, 2, 3, 4\}$ and the branches that we have taken to get to the point where we are are $x \neq 1$ and $x \neq 2$. The generated new models will all have the constraints $x \neq 1$ and $x \neq 2$ to get to the point in the search tree where we split the problem. Then we add constraints to partition the search space based on the remaining values in the domain of x similar to the previous example.

Using this technique, we can create new chunks of work whenever a worker becomes idle by simply asking one of the busy workers to stop and generate split models. The search is then resumed from where it was stopped and the remaining search space is explored in parallel by the two workers. Note that there is a runtime overhead involved with stopping and resuming search because the constraints which enable resumption must be propagated and the solver needs to explore a small number of search nodes to get to the point where it was stopped before. There is also a memory overhead because the additional constraints need to be stored.

We have implemented this approach in a development version of Minion, which we are planning to release to the public after further testing and verifi-

[2] This same technique allows Minion to be paused and resumed: the nogoods are provided when the solver is interrupted, and can be used to restart search, potentially using a different solver, different search strategy or on a different machine.

cation. Initial experiments showed that the overhead of stopping, splitting and resuming is minimal and not significant for large problems.

In practice, we run Minion for a specified amount of time, then stop, split and resume instead of splitting at the beginning and when workers become idle. The algorithm is detailed in Figure 1. This creates an n-ary split tree of models for n new models generated at each split. Initially, the potential for distribution is small but grows exponentially as more and more search is performed.

```
Input  : constraint problem $X$, allotted time $T_{max}$ and splitting factor $n \geq 2$
Output: a solution to $X$ or nothing if no solution has been found
run Minion with input $X$ until termination or $T_{max}$;
if solved?($X$) then
    terminate workers;
    return solution;
else if search space exhausted? then
    return;
else
    $X' \leftarrow X$ with new constraints ruling out search already performed;
    split $X'$ into $n$ parts $X'_1, \ldots, X'_n$;
    for $i \leftarrow 1$ to $n$ do in parallel
        distSolve($X'_n$, $T_{max}$,$n$);
    end
end
```

Fig. 1. distSolve(X,T_{max},n): Recursive procedure to find the first solution to a constraint problem distributed across several workers.

4 Comparison to existing approaches

We see the main advantage of our approach in not requiring any involved changes to the constraint solving system to support distributed solving; in particular communications between workers. Conventionally, distribution is achieved with the aid of recomputation and cloning; established techniques used e.g. in [16]. We require two features of our solver: partitioning using constraints, and ability to output restart nogoods. Our system makes use of *cloning*, which we call "splitting" and implement by means of nogoods added to the constraint model in order to partition the domain of a variable. However, where other systems use *recomputation*, our system uses restart nogoods. In a system based on recomputation the clone begins at specific search path, e.g. stolen from another worker; with restart nogoods notionally multiple search paths are provided and the solver may explore these in any way it wishes, not necessarily one after the other. It is merely a convenient and compact way of encoding the situation where a solver is relinquishing *all* its remaining work.

Contemporary constraint solvers make it easy to change or amend the search procedure to support distribution across several executors, but even then changes to the constraint solving system are required. While an initial implementation of distributed search can be done relatively quickly, handling failure properly and supporting things like nodes being added and removed dynamically requires significantly more effort. Our approach separates this part completely from the constraint solving system.

There are several advantages to implementing distributed solving the way given in Figure 1. First, by creating regular "snapshots" of the search done, the resilience against failure increases. Every time we stop, split and resume, the modified models are saved. As they contain constraints that rule out the search already done, we only lose the work done after that point if a worker fails. This means that the maximum amount of work we lose in case of a total failure of all workers is the allotted time T_{max} times the number of workers $|w|$.

The fact that the modified models can be stored can also be exploited to move the solving process to a different set of workers after it has been started without losing any work. It furthermore means that we require no communication between the individual workers solving the problem; they only need to be able to receive the problem to solve and send the solution or split models back.

Another advantage is that small problems which Minion can solve within the allotted time are not split and no distribution overhead is incurred. Solving proceeds as it would in a standard, non-distributed fashion.

Our approach is particularly suitable for use with existing grid-computing software or workload management systems such as Condor[3]. Every time new models are generated, they are submitted to the system which queues them and allocates a worker as soon as one becomes available. By leveraging existing software to perform this task, a huge amount of development time is saved and errors are avoided. For large problems, the number of queued jobs will usually exceed the number of workers, ensuring good resource utilisation.

The management system to monitor the search, queue split models and terminate the workers if a solution has been found can be implemented efficiently in just a few lines of code. We have written a Ruby script that performs this task in little more than an hour. Obviously there is potential for trying different search strategies for different branches or modifying other search parameters in order to improve efficiency. With the appropriate modifications, the management system could adapt the search procedure specifically for individual parts of the search tree. We are planning to explore these possibilities in future work.

A downside of the approach is that the number of models which can be solved in parallel will be small to start with. This means that the utilisation of resources in the beginning will be suboptimal. Only as more and more search space is explored and more and more split models are generated, the utilisation will improve. This however can be mitigated by dynamically adapting the time

[3] http://www.cs.wisc.edu/condor/

for which Minion is run before splitting the problem – in the beginning, we set it to a small value to quickly get many models that we can solve in parallel. Then we gradually increase the allotted time as the resource utilisation improves.

Our technique is intended to be used for very large problems which take a long time (many hours, days or weeks) to solve. It is unlikely to be efficient for problems that can be solved in minutes, but on the other hand there is no need for distributed solving if the problem can be solved sequentially in a short amount of time. Only large search spaces can be split in a way that many workers are kept busy without a high communication overhead.

5 Detailed example

We will now have a detailed look at how our approach works for a specific problem. Consider the 4-queens problem. We want to place 4 queens on a 4×4 chessboard such that no queen is attacking another queen. Queens can move along rows, columns and diagonals. The constraints therefore have to forbid that two or more queens are in the same row, the same column or on the same diagonal. The constraint model in Figure 2 captures this problem.

```
language Dominion 0.1
letting n = 4
dim queens[n]: int
find queens[..]: int {1..n}
such that
alldifferent alldiff(queens[..])
diagonals1 [ not(eq1 eq(queens[i], add(queens[j], j-i))) |
    i in {0..n-2}, j in {i+1..n-1} ]
diagonals2 [ not(eq2 eq(queens[i], add(queens[j], i-j))) |
    i in {0..n-2}, j in {i+1..n-1} ]
```

Fig. 2. Model for the 4-queens problem in the Dominion language [4]. The model describes the n-queens problem in general and is specialised for 4-queens in the second line.

We assume variable ordering $queens_0, queens_1, queens_2, queens_3$, ascending value ordering from 1 to 4 and n-way branching. The search tree for a simple backtracking algorithm is depicted in Figure 3. Even for a very small problem like this, there is significant potential for distributed solving.

We now start solving the problem until we reach the assignment $queens_0 = 2$. Then we stop. The constraint we need to add to resume the search at the same point is

```
resume not(innerresume eq(queens[0], 1))
```

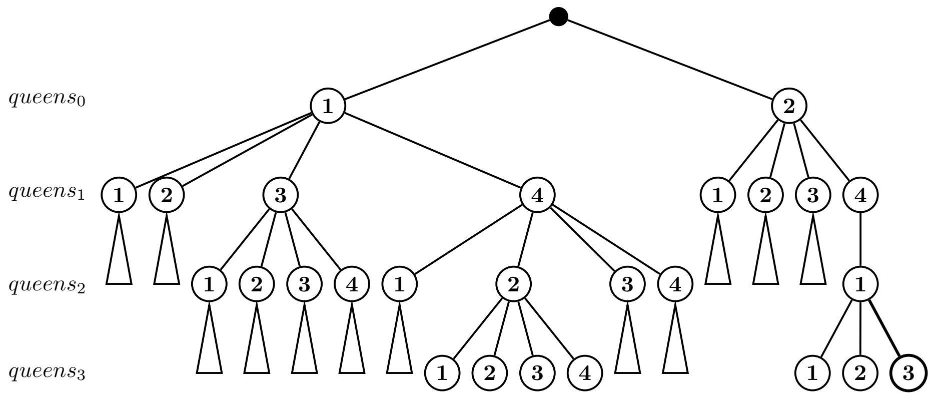

Fig. 3. First solution search tree for 4-queens. The triangles depict subtrees which are not explored because the partial assignment so far cannot be part of a solution. The bold, rightmost node is where the solution is found. The levels of the tree show assignments to the variables shown on the left.

(note that `resume` and `innerresume` are simply identifiers given to the specific constraints as required in the Dominion language [4]).

Let us assume a splitting factor of 2. We add the constraints to split the remaining search space as follows. The variable currently under consideration is $queens_1$, its domain is $\{1, 2, 3, 4\}$ and therefore the constraints are

`left leq(queens[1], 2)` and `right leq(3, queens[1])`.

The search is restarted with two workers, each exploring separate branches of the remaining search space. The first worker finds no solutions in its part of the search space, terminates and returns. The second worker finds a solution and returns it. Search terminates and no further splitting is performed.

6 Conclusions and future work

We have proposed and detailed a novel approach for distributing constraint problems across multiple computers. Instead of modifying the solver to support distributed operation, we only require some simple and generic modifications that post additional constraints to the model.

The main advantages of our approach are that it does not require networked machines, is resilient against failure and can be implemented easily in constraint solvers which are aware of the state of the search.

The main drawback of this paper is that we do not have performed a systematic experimental evaluation of our approach yet. In the future, we would like to evaluate it in terms of solving speedup and resource utilisation on large, real-world problems. Furthermore, we would like to investigate finding all solutions to a constraint problem and solving constrained optimisation problems in a distributed manner.

Adapting the search procedure and parameters dynamically during search is another promising area for future work. The solving process could be tailored to the characteristics of parts of the search space to improve efficiency.

Another direction for future work is to support a higher level of abstraction for decomposing problems into subproblems. This would be necessary to support problems which cannot be decomposed by simply adding constraints that split the domain of a variable.

Acknowledgements

The authors thank Chris Jefferson for help with implementing the model splitting in Minion and the anonymous reviewers for their feedback. Lars Kotthoff is supported by a SICSA studentship.

References

1. Blumofe, R.D., Leiserson, C.E.: Scheduling multithreaded computations by work stealing. J. ACM 46(5), 720–748 (1999)
2. Collin, Z., Dechter, R., Katz, S.: On the feasibility of distributed constraint satisfaction. In: IJCAI. pp. 318–324 (1991)
3. Ezzahir, R., Bessiere, C., Belaissaoui, M., Bouyakhf, H., Mohammed, U., Agdal, V.: DisChoco: A platform for distributed constraint programming (2007)
4. Gent, I.P., Jefferson, C., Kotthoff, L., Miguel, I., Nightingale, P.: The DOMINION input language version 0.1. CIRCA preprint 2009/21, University of St Andrews (2009), http://www-circa.mcs.st-and.ac.uk/Preprints/InLangSpec.pdf
5. Gent, I.P., Jefferson, C., Miguel, I.: MINION: a fast, scalable, constraint solver. In: ECAI. pp. 98–102 (2006)
6. Hamadi, Y.: Disolver 3.0: the Distributed Constraint Solver version 3.0 (2007)
7. Kumar, V., Rao, V.N.: Parallel depth first search. Part II. analysis. Int. J. Parallel Program. 16(6), 501–519 (1987)
8. Lecoutre, C., Sais, L., Tabary, S., Vidal, V.: Nogood recording from restarts. In: IJCAI'07: Proceedings of the 20th international joint conference on Artifical intelligence. pp. 131–136. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2007)
9. Michel, L., Hentenryck, P.V.: A decomposition-based implementation of search strategies. ACM Trans. Comput. Logic 5(2), 351–383 (2004)
10. Michel, L., See, A., Hentenryck, P.V.: Parallelizing constraint programs transparently. In: CP. pp. 514–528 (2007)
11. Petcu, A.: FRODO: a FRamework for Open/Distributed constraint optimization. Technical report no. 2006/001, Swiss Federal Institute of Technology (EPFL) (2006), http://liawww.epfl.ch/frodo/
12. Rao, V.N., Kumar, V.: On the efficiency of parallel backtracking. IEEE Trans. Parallel Distrib. Syst. 4(4), 427–437 (1993)
13. Rao, V.N., Kumar, V.: Parallel depth first search. Part I. implementation. Int. J. Parallel Program. 16(6), 479–499 (1987)
14. Rolf, C.C., Kuchcinski, K.: Load-balancing methods for parallel and distributed constraint solving. In: CLUSTER. pp. 304–309 (2008)

15. Sanders, P.: Better algorithms for parallel backtracking. In: Workshop on Algorithms for Irregularly Structured Problems. pp. 333–347 (1995)
16. Schulte, C.: Parallel search made simple. In: Proceedings of TRICS. pp. 41–57 (2000)
17. Yokoo, M., Durfee, E.H., Ishida, T., Kuwabara, K.: Distributed constraint satisfaction for formalizing distributed problem solving. In: 12th IEEE International Conference on Distributed Computing Systems. pp. 614–621 (1992)
18. Yokoo, M., Durfee, E.H., Ishida, T., Kuwabara, K.: The distributed constraint satisfaction problem: Formalization and algorithms. IEEE Trans. on Knowl. and Data Eng. 10(5), 673–685 (1998)

System Demonstration

GenDebugger: An Explanation-based Constraint Debugger[*]

Giora Alexandron[1], Vitaly Lagoon[1], Reuven Naveh[1], and Aaron Rich[1],

[1] Cadence Design Systems, Inc.
8 Hamelacha St., Rosh Ha'ain, 48091, Israel
{giora, lagoon, rnaveh, ari}@cadence.com

Simulation-based verification is a technique used to ensure correctness of hardware designs. To verify a design test scenarios are generated randomly based on architectural and test dependencies modeled by constraints. This method is supported by verification-oriented languages such as ***e*** [3,7] and System Verilog [1,4]. Environments based on constraints may suffer from coding errors and thus require debugging tools. Coding errors in constraints are revealed in several forms, such as failure to find a solution, unexpected values assigned to variables, and unexpected distribution of values across a set of simulations. Bad runtime and memory consumption of constraint solving activity is also a common problem.

A tool generating random scenarios for a verification environment is required to meet several requirements. First and foremost, it must find a proper solution satisfying the constraints. Second, it must be able to find multiple solutions of a problem, such that they are properly distributed within the entire solution space. Lastly, as generation is one of the most time-consuming operations in the verification process, it must be efficient. The debugging tool needs to address cases in which these requirements are not fulfilled: either no solution is found, or the solution does not meet user-expectations, or it takes too long to come up with a solution.

Constraint debugging poses a considerable challenge. Existing approaches, including adding/removing constraints, tracing of the solving process [5], or printing constraints participating in a conflict, are insufficient in complex scenarios. Debugging tools such as source line debuggers, with which most software engineers are familiar, are ill-suited for debugging constraints. The reason is that source-line debuggers are normally sequential, showing and executing the line-by-line imperative flow of user code. Constraints, in contrast, are declarative entities.

In this work we describe GenDebugger, the generation debugger for Specman [7]. Specman is an Electonic Design Automation (EDA) tool that provides advanced constraint-based functional verification of hardware designs. GenDebugger is a

[*] This work is a revised version of a paper presented at the Haifa Verification Conference in 2009 [6].

component of IntelliGen, the new generator of Specman. GenDebugger has been used successfully for over three years, and received positive feedback from its customers.

GenDebugger is an explanation-based constraint debugger [2]. While some principles of GenDebugger are applicable to any solving technique, its sequential depiction of the generation process best fits propagation-based solvers, such as IntelliGen.

GenDebugger depicts the constraint-solving process as a sequence of elementary *solving steps*. The main three kinds of solving steps are assignment of a value to a variable, reduction of one or more variable domains through propagation, and backtracking from a previous assignment due to a conflict. There are additional kinds of steps specific to IntelliGen, corresponding to application/withdrawal of soft constraints, application of distribution policies, etc. The collection of all solving steps is organized in a *search tree* which can be inspected and explored using the rich set of GUI tools.

GenDebugger shows each step in a detailed, interactive view. The variables and constraints involved in each step are displayed so that navigation between relevant kinds of information is quick. The tool displays both the initial and the resulting domains for steps involving domain reduction. Any variable or constraint can be selected or queried. For each variable we present the list of steps which modified its domain, thus giving the *explanation*. The variables are organized in two different panes, one showing them according to the chronological order in which they were assigned, and another one showing the ‘generation tree’, i.e. their hierarchical positions in the verification environment.

GenDebugger can be invoked in various ways. The process of generation can either be stopped at a point of conflict, or at the generation of a specific variable, or at the beginning of the generation process. Additionally, the process can be manually interrupted by the user and resumed in GenDebugger. The generation itself can be debugged using either a step-by-step process or retrospectively. By enabling the user to debug generation as it occurs, GenDebugger gives its users a feeling similar to debugging of procedural code.

In this talk we demonstrate the GenDebugger GUI and the concepts of explanation-based constraint debugging. We show on a set of simple examples how the various types of bugs in constraints can be efficiently diagnosed using GenDebugger.

References

1. Bergeron, J., Cerny, E., Hunter, A., and Nightingale, A. 2005 Verification Methodology Manual for Systemverilog. Springer-Verlag New York, Inc.
2. Ghoniem M., Jussien N. and Fekete J. D.: VISEXP: Visualizing Constraint Solver Dynamics Using Explanations. In FLAIRS'04: Seventeenth International Florida Artificial Intelligence Research Society Conference. AAAI press, Miami Beach, FL (2004)
3. IEEE Standard for the Functional Verification Language 'e', IEEE Computer Society, IEEE,

New York, NY, IEEE Std 1647 (2006)
4. IEEE Standard For System Verilog - Unified Hardware Design, Specification and Verification Language, IEEE Computer Society, IEEE, New York, NY, IEEE Std 1800 (2005)
5. Meier M.: Debugging Constraint Programs. In V. Saraswat and P.V. Hentenryck (Eds.): Principles and Practice of Constraint Programming, pp. 204-221. Lecture Notes in Computer Science 976. MIT (1995)
6. Rich A., Alexandron G. and Naveh R.: An Explanation-Based Constraint Debugger. HVC 2009, Haifa, Israel (October 19-22, 2009)
7. Robinson, D.: Aspect-Oriented Programming with the e Verification Language: A Pragmatic Guide for Testbench Developers. Elsevier Inc.

Combining Parallel Search and Parallel Consistency in Constraint Programming

Carl Christian Rolf and Krzysztof Kuchcinski

Department of Computer Science, Lund University, Sweden
`Carl_Christian.Rolf@cs.lth.se`, `Krzysztof.Kuchcinski@cs.lth.se`

Abstract. Program parallelization becomes increasingly important when new multi-core architectures provide ways to improve performance. One of the greatest challenges of this development lies in programming parallel applications. Declarative languages, such as constraint programming, can make the transition to parallelism easier by hiding the parallelization details in a framework.
Automatic parallelization in constraint programming has mostly focused on parallel search. While search and consistency are intrinsically linked, the consistency part of the solving process is often more time-consuming. We have previously looked at parallel consistency and found it to be quite promising. In this paper we investigate how to combine parallel search with parallel consistency. We evaluate which problems are suitable and which are not. Our results show that parallelizing the entire solving process in constraint programming is a major challenge as parallel search and parallel consistency typically suit different types of problems.

1 Introduction

In this paper, we discuss the combination of parallel search and parallel consistency in constraint programming (CP). CP has the advantage of being declarative. Hence, the programmer does not have to make any significant changes to the program in order to solve it using parallelism. This means that the difficult aspects of parallel programming can be left entirely to the creator of the constraint framework.

Constraint programming has been used with great success to tackle different instances of NP-complete problems such as graph coloring, satisfiability (SAT), and scheduling [1]. A constraint satisfaction problem (CSP) can be defined as a 3-tuple $P = (X, D, C)$, where X is a set of variables, D is a set of finite domains where D_i is the domain of X_i, and C is a set of primitive or global constraints containing several of the variables in X. Solving a CSP means finding assignments to X such that the value of X_i is in D_i, while all the constraints are satisfied. The tuple P is referred to as a constraint store.

Finding a valid assignment to a constraint satisfaction problem is usually accomplished by combining backtracking search with consistency checking that prunes inconsistent values. In every node of the search tree, a variable is assigned one of the values from its domain. Due to time-complexity issues, the consistency methods are rarely complete [2]. Hence, the domains will contain values that are locally consistent, i.e., they will not be part of a solution, but we cannot prove this yet.

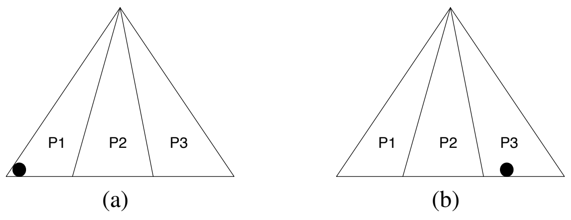

Fig. 1. The position of the solution in a search tree affects the benefit of parallelism.

The examples in Fig. 1 illustrates the problem of parallelism in CP. We use three processors: P1, P2, and P3 to find the solution. We assign the different parts of the search tree to processors as in the figure. The solution we are searching for is in the leftmost part of the search tree in Fig. 1(a) and will be found by processor P1. Any work performed by processor P2 and P3 will therefore prove unnecessary and will only have added communication overhead. In this case, using P2 and P3 for parallel consistency will be much more fruitful. On the other hand, in Fig. 1(b), the solution is in the rightmost part of the tree. Hence, parallel search can reduce the total amount of nodes explored to less than a third. In this situation, parallel consistency can still be used to further increase the performance.

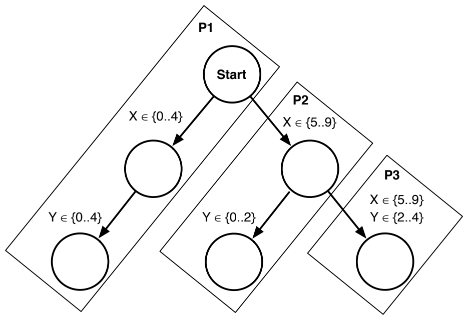

Fig. 2. Parallel search in constraint programming.

In this paper, we will refer to parallel search (OR-parallelism) as data parallelism, and parallel consistency (AND-parallelism) as task parallelism. Parallelizing search in CP can be done by splitting data between solvers, e.g., create a decision point for a selected variable X_i so that one computer handles $X_i < \frac{min(X_i)+max(X_i)}{2}$ and another handles $X_i \geq \frac{min(X_i)+max(X_i)}{2}$. An example of such data parallelism in CP is depicted in Fig. 2. The different possible assignments are explored by processors P1, P2, and P3. Clearly, we are not fully utilizing all three processors in this example. At the first level of the search tree, only two out of three processors are active. Near the leafs of the search tree, communication cost outweighs the benefit of parallelism. Hence, we often have a low processor load in later part of the search.

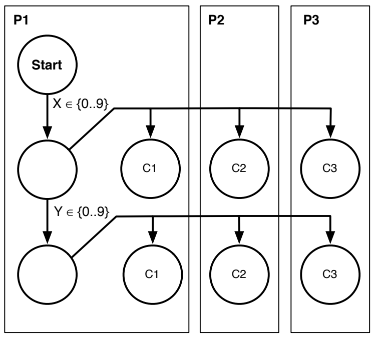

Fig. 3. Parallel consistency in constraint programming.

Figure 3 presents the model of parallel consistency in constraint programming which we will partly discus in this paper. In the example, the search process is sequential, but the enforcement of consistency is performed in parallel. Constraints C1, C2, and C3 can be evaluated independently of each other on different processors, as long as their pruning is synchronized. We do not share data during the pruning, hence, we may have to perform extra iterations of consistency. The cause of this implicit data dependency is that global constraint often rely on internal data-structures that become incoherent if variables are modified during consistency.

The problem of idle processors during the latter parts of the search is pervasive [3, 4]. Regardless of the problem, the communication cost will eventually become too big.

Data parallelism can be problematic, or even unsuitable, for other reasons. Many problems modeled in CP spend a magnitude more time enforcing consistency than searching. Using data parallelism for these problems often reduces performance. In these cases, task parallelism is the only way to take advantage of multicore processors.

By combining parallel consistency with parallel search, we can further boost the performance of constraint programming.

The rest of this paper is organized as follows. In Section 2 the background issues are explained, in Section 3 the parallel consistency is described. Section 4 details how we combine parallel search and parallel consistency. Section 5 describes the experiments and the results, Section 6 gathers our conclusions.

2 Background

Most work on parallelism in CP has dealt with parallel search [5, 6]. While this offers the greatest theoretical scalability, it is often limited by a number of issues. Today, the main one is that processing disjoint data will saturate the memory bus faster than when processing the same data. In theory, a super-linear performance should be possible for depth-first search algorithms [7]. This, however, has only rarely been reported, and only for small numbers of processors [6]. The performance-limits of data parallelism in memory intense applications, such as CP, are especially apparent on modern multi-core architectures [8].

Task parallelism is the most realistic type of parallelism for problems where the time needed for search is insignificant compared to that of enforcing consistency. This happens when the consistency algorithms prune almost all of the inconsistent values. Such strong pruning is particularly expensive and in a greater need of parallelism. The advantage of these large constraints over a massively parallel search is that the execution time may become more predictable. For instance, speed-up when searching for one solution often has a high variance when parallelizing search since the performance is highly dependent on which domains are split.

Previous work on parallel enforcement of consistency has mostly focused on parallel arc-consistency algorithms [9, 10]. The downside of such an approach is that processing one constraint at a time may not allow inconsistencies to be discovered as quickly as when processing many constraints in parallel. If one constraint holds and another does not, the enforcement of the first one can be cancelled as soon as the inconsistency of the second constraint is discovered.

The greatest downside of parallel arc-consistency is that it is not applicable to global constraints. These global constraints encompass several, or all, of the variables in a problem. This allows them to achieve a much better pruning than primitive constraints, which can only establish simple relations between variables, such as $X + Y \leq Z$.

We only know of one paper on parallel consistency with global constraints [11]. That paper reported a speed-up for problems that can be modeled so that load-balancing is not a big issue. For example, Sudoku gave a near-linear speed-up. However, in this paper we go further by looking at combining parallel search with parallel consistency.

3 Parallel Consistency

Parallel consistency in CP means that several constraints will be evaluated in parallel. Constraints that contain the same variables have data dependencies, and therefore their pruning must be synchronized. However, since the pruning is monotonic, the order in which the data is modified does not affect the correctness. This follows from the property that well-behaved constraint propagators must be both decreasing and monotonic [12]. In our finite domain solver this is guaranteed since the implementation makes the intersection of the old domain and the one given by the consistency algorithm. The result is written back as a new domain. Hence, the domain size will never increase.

Our model of parallel consistency is depicted in Fig. 4, this model is described in greater detail in [11] and in Fig. 6(b). At each level of the search, consistency is enforced. This is done by waking the consistency threads available to the constraint program. These threads will then retrieve work from the queue of constraints whose variables have changed. In order to reduce synchronization, each thread will take several constraints out of the queue at the same time. When all the constraints that were in the queue at the beginning of the consistency phase have been processed, all prunings are committed to the constraint store as the solver performs updates. If there were no changes to any variable, the consistency has reached a fix-point and the constraint program resumes the search. If an inconsistency is discovered, the other consistency threads are notified and they all enter the waiting state after informing the constraint program that it needs to backtrack.

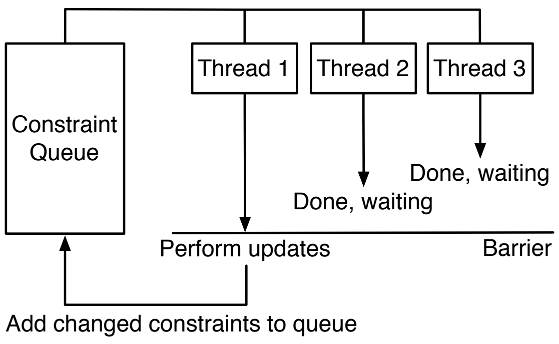

Fig. 4. The execution model for parallel consistency.

Consistency enforcement is iterative. When the threads are ready, the constraint queue is split between them, and one iteration of consistency can begin. This procedure will be repeated until we reach a fixpoint, i.e., the constraints no longer change the domain of any variable. The constraints containing variables that have changes will be added to the constraint queue after the updates have been performed.

One of the main concerns in parallel consistency is visibility. Global constraints usually maintain an internal state that may become incoherent if some variables are changed while the consistency algorithm is running. If we perform the pruning in parallel, the changes will only be visible to the other constraints after the barrier. This reduces the pruning achieved per consistency iteration. Hence, in parallel consistency, we will usually perform several more iterations than in sequential consistency before we reach the fixpoint.

4 Combining Parallel Search and Parallel Consistency

The idea when combining parallel search and parallel consistency is to associate every search thread several consistency threads. A simple example is depicted in Fig. 5. First the data is split from processor P1 and sent to processor P2. Then the search running on P1 will perform consistency by evaluating constraints C1 and C2 on processors P1 and P3 respectively. The search running on P2 will, completely independently, run consistency using processors P2 and P4. Each search has its own store, hence, constraints C1 and C2 can be evaluated by the two searches without any synchronization.

More formally, the execution of the combined search and consistency in CP proceeds as follows. We begin with a constraint store $P = (X, D, C)$ as defined earlier. This gives us a search space to explore, which can be represented as a tree. The children of the root node represent the values in D_i. In these nodes, we assign X_i one of its possible values and remove X_i from X. For example, assigning X_0 the value 5 gives a node n with $P_n = (X \setminus X_0, D \cup D_0 \cap \{5\}, C)$. After each assignment, we apply the the function $enforceConsistency$, which runs the consistency methods of C, changing our store to (X', D', C) where $X' = X \setminus X_i$. D' is the set of finite domains representing the possible values for X' that were not marked as impossible by the consistency

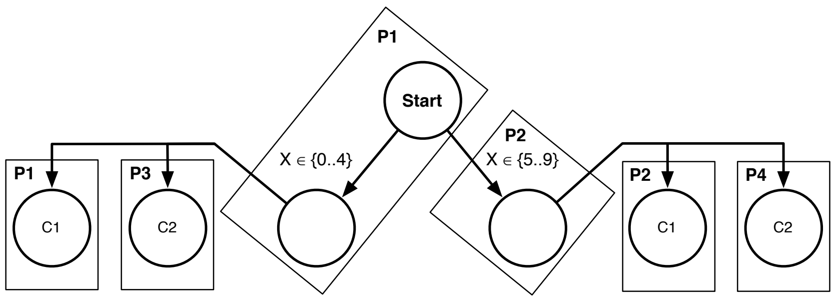

Fig. 5. An example of combining parallel search and parallel consistency.

methods of C. The method $enforceConsistency$ is applied iteratively until $D'' = D'$. Now there are two possibilities: either $\exists D'_i = \emptyset$, in which case we have a failure, meaning that there are no solutions reachable from this node, or we progress with the search. In the latter case, we have two sub-states. Either $X' = \emptyset$, in which case we have found a solution, or we need to continue recursively by picking a new X_i.

Parallel search means that we divide D_i into subsets and assign them to different processors. Each branch of the search tree starting in a node is independent of all other branches. Hence, there is no data dependency between the different parts of the search space. *Parallel consistency* means parallelizing the $enforceConsistency$ method. This is achieved by partitioning C into subsets, each handled by a different processor.

The pseudo code for our model is presented in Fig. 6. When a search thread makes an assignment it needs to perform consistency before progressing to the next level in the search tree. Hence, processors P1 and P2 in the example are available to aid in the consistency enforcement. The consistency threads are idle while the search thread works. If we only allocate one consistency thread per processor a lot of processors will be idle as we are waiting to perform the assignment. Hence, it is a good idea to make sure that the total number of consistency threads exceeds the number of processors.

As Fig. 6 shows, the parallel search threads will remove a search node and explore it. In our model, a search node represents a set of possible values for a variable. The thread that removes this set guarantees that all values will be explored. If the set is very large, the search thread can split the set to allow other threads to aid in the exploration. When there are no more search nodes to explore, the entire search space has been explored.

Since we have to wait for the different threads, some parts of the algorithm are, by necessity, synchronized. In Fig. 6(a), line 15 requires synchronization while we wait for the consistency threads to finish. In Fig. 6(b), lines 15 to 22, which represent the barrier, are synchronized. However, each thread may use its own lock for waiting. Hence, there is little lock contention. Furthermore, line 13 has to be synchronized in order to halt the other threads when we have discovered an inconsistency. Depending on the data structure, lines 6 and 7 may also have to be synchronized.

```
// search nodes to be explored $N$
// variables to be labeled $V$, with FDV $x_i \in V$
// domain of $x_i$ is $d_i$, list of slave computers $S$

while $N \neq \emptyset$
    $Node \leftarrow N.first$
    $N \leftarrow N \setminus Node$
    $V \leftarrow Node.unlabeledVariables$
    while $V \neq \emptyset$
        $V \leftarrow V \setminus x_i$
        select value $a$ from $d_i$
        $x_i \leftarrow a$
        for each slave $s$ in $S$
            $s$.enforceConsistency
        wait  //wait for all slaves to stop
        if Inconsistent
            $d_i \leftarrow d_i \setminus a$
            $V \leftarrow V \cup x_i$
    end while
    store solution
end while
```

(a)

```
// set of constraints to be processed $PC$
// set of constraints processed in this slave $SC$
// returns result to the constraint program

boolean enforceConsistency
    while $PC \neq \emptyset$
        $PC \leftarrow PC \setminus SC$
        while $SC \neq \emptyset$
            $SC \leftarrow SC \setminus c$
            $c$.consistency
            if $c$.inconsistent
                for each slave $s$ in $S$
                    $s$.stop
                return Inconsistent
            if all other slaves waiting
                perform updates
                for each changed constraint $cd$
                    $PC \leftarrow PC \cup cd$
                for each slave $s$ in $S$
                    $s$.wake
            else
                wait  //wait for updates
        end while
    end while
    return Consistent
```

(b)

Fig. 6. The combined parallel search and parallel consistency algorithm. Parallel depth-first search (a), slave program for parallel consistency (b).

4.1 Discussion

An alternative way to combine parallel search and consistency is to use a shared work-queue for both types of jobs. Threads that become idle could get new work from the queue, whether it was running consistency for a constraint or exploring a search space. However, the performance of such an approach would be heavily dependent on the priority given to the different types of work. If the priorities were just slightly incorrect, it would hurt the performance of the other threads. For instance, a thread wanting help with consistency might never get it because the idle threads are picking up search jobs instead. It might be possible to solve this problem using adaptive priorities. However, this is outside the scope of this paper

By combining parallel search and parallel consistency we hope to achieve a better scalability. Unlike data parallelism for depth-first search, the splitting of data poses a problem in constraint programming. The reason is that the split will affect the domains of the variables that have not yet been assigned a value. In the example in Fig. 2, with a constraint such as $X > Y$ the consistency will change the shape of the search tree by removing the value 4 from the domain of Y for processor P1. For more complex problems, the shape of both search trees may be affected in unpredictable ways. Since the consistency methods are not complete, there is no way to efficiently estimate the size and shape of the search trees after a split. Parallel consistency allows us to use the hardware more efficiently when parallel search runs into these kinds of problems.

In [11] we showed that parallel consistency scales best on very large problems consisting of many global constraints. Solving such problems is a daunting task, which makes it hard to combine parallel search with parallel consistency. Furthermore, finding just one solution to a problem often leads to non-deterministic speed-ups.

The biggest obstacle we faced when developing a scalable version of parallel consistency was the cost of synchronization. The problem comes from global constraints, these typically use internal data structures. For instance, the bounds consistency for

AllDifferent constraint uses a list where the order of variables is given by Hall intervals [13]. If pruning is performed instantly by other threads, instead of being stored until a barrier, the integrity of these data structures may be compromised. Eliminating barrier synchronization would greatly increase the performance of parallel consistency.

5 Experimental Results

We used the JaCoP solver [14] in our experiments. The experiments were run on a Mac Pro with two 3.2 GHz quad-core Intel Xeon processors running Mac OS X 10.6.2 with Java 6. These two processors have a common cache and memory bus for each of its four cores. The parallel version of our solver is described in detail in [3].

5.1 Experiment Setup

We used two problems in our experiments: n-Sudoku, which gives an $n \times n$ Sudoku if the square root of n is an integer and n-Queens which consists in finding a placement of n queens on a chessboard so that no queen can strike another. Both problems use the AllDifferent constraint with bounds consistency [13], chosen since it is the global constraint most well spread in constraint solvers. The characteristics of the problems are presented in Table 1.

The results are the absolute speed-ups when searching for a limited number of solutions to n-Sudoku and one solution to n-Queens. For Sudoku we used $n = 100$ with 85 % of values set and searched for 200 and 5 000 solutions. For Queens we used $n = 550$ and searched for a single solution. We picked these problems in order to illustrate how the size of the search space affects the behavior when combining parallel search with parallel consistency, while still having a reasonable execution time.

For each problem we used between one and eight search threads. For each search thread we used between one and eight consistency threads. We used depth-first search with in-order variable selection for both problems. The sequential performance of our solver is lower than that of some others. However, this overhead largely comes from the higher memory usage of a Java based solver. On a multicore system this is a downside since the memory bus is shared. Hence, lower sequential performance does not necessarily make it simpler to achieve a high speed-up.

Table 1. Characteristics of the problems.

Problem	Variables	Primitive Constraints	Global Constraints
Sudoku	10 000	0	300
Queens	1 648	1 098	3

5.2 Results for Sudoku

The results for 100-Sudoku is presented in Table 2 and Table 3, the speed-ups are depicted in Fig. 7 and Fig. 8. The bold number in the table indicates the fastest time and the gray background marks the times slower than sequential. The results show that there is a clear difference in behavior as the search space increases. When we have to explore a larger search space, parallel search is better than parallel consistency. However, if we have a more even balance between search and consistency, combining the two types of parallelism increases the performance.

From the diagrams, we can see that it is good to use more consistency threads than there are processor cores. However, using many more threads is not beneficial, especially when there are several search threads.

It is noteworthy that there is little overhead for using parallel consistency when only running one search thread. Search for 200 solutions even increases the performance somewhat. This is important because it means that parallel consistency can be successful when it is difficult to extract data parallelism from the problem.

The reduction in performance when adding parallel consistency to the search for 5 000 solutions comes to some extent from synchronization costs. Synchronization in Java automatically invalidates cache lines that may contain data useful to other threads. With more precise control over cache invalidation, the execution time overhead added by the parallel consistency can be reduced.

Using too many threads will cause an undesirable amount of task switching and saturation of the memory bus. We measured and analyzed how the number of active threads, and their type, affects performance. The average number of active threads when running two search threads and four consistency threads per search thread for 200 solutions to n-Sudoku was 5.5. This is the average over the entire execution time. The same number for the slowest instance, eight by eight threads, was 59 active threads. The first case achieves a rather good balance given that it is hard to extract useful data parallelism for the search threads. The number of active threads for the search threads alone was 1.5 when using two search threads, and 7.1 when using eight search threads.

Table 2. Execution times in seconds when searching for 200 solutions to 100-Sudoku.

Consistency Threads	Search Threads			
per Search Thread	1	2	4	8
1	176	125	122	145
2	176	124	143	177
4	158	**110**	142	210
8	162	127	192	269

The main bottleneck for the performance is the increased workload to enforce consistency. The total number of times constraints are evaluated per explored search node is depicted in Fig. 9 and Fig. 10. Clearly, using parallel consistency increases the number of times we have to evaluate the constraints. This is because we cannot share data between constraints during their execution.

Table 3. Execution times in seconds when searching for 5 000 solutions to 100-Sudoku.

Consistency Threads per Search Thread	Search Threads 1	2	4	8
1	3 663	1 882	1 720	**1 649**
2	3 931	2 293	2 565	2 782
4	3 995	2 161	3 224	2 735
8	4 254	2 556	3 997	3 192

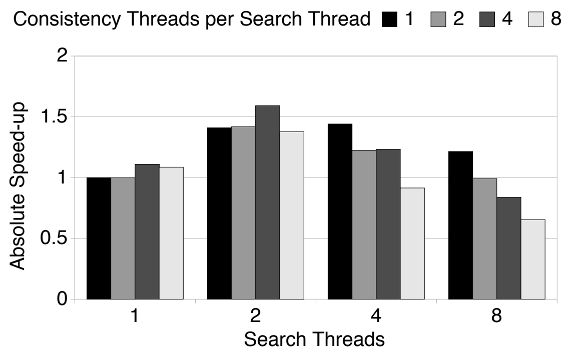

Fig. 7. Speed-up when searching for 200 solutions to 100-Sudoku.

The second bottleneck for the performance of parallel consistency is synchronization. In our solution, we have several points of synchronization. The barrier before updates is particularly costly as the slowest consistency thread determines the speed.

The third bottleneck is the speed of the memory bus. Parallel search can quickly saturate the bus. Adding parallel consistency will worsen the performance. The performance clearly drops off towards the lower right hand corner of Table 2 and to the left of Table 3. This problem can to some extent be avoided by having a shared queue of tasks and a fix amount of threads in the program. These threads could then switch between performing consistency and search in order to adapt to the memory bus load.

The only way to fruitfully combine parallel search with parallel consistency is if we reduce the number of search nodes more than we increase their computational weight. The inherent problem in doing this is clear from the differences in results between Table 4 and Table 5. As shown by Fig. 9, when the problem is small there is an almost linear increase in the number of consistency checks per search node as we add search threads. On the other hand, Fig. 10 shows that the number of consistency checks varies a lot depending on the number of consistency threads. The reason is that when we have to explore a large search space we will run into more inconsistencies, which can be detected faster when using parallel consistency. However, inconsistent nodes have less computational weight. In conclusion, when parallel search starts to become useful, parallel consistency cannot pay off the computational overhead it causes.

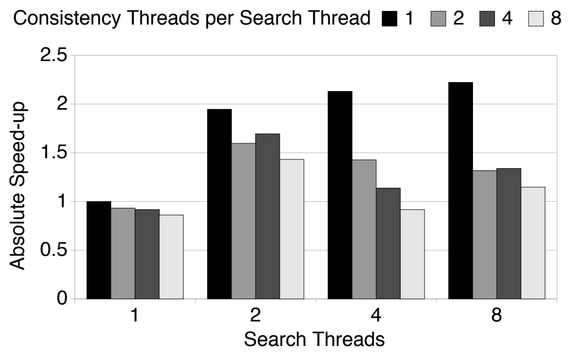

Fig. 8. Speed-up when searching for 5 000 solutions to 100-Sudoku.

Table 4. Number of times consistency was called for the constraints in 100-Sudoku when searching for 200 solutions.

Consistency Threads	Search Threads			
per Search Thread	1	2	4	8
1	23 475	47 487	122 587	217 339
2	36 585	73 017	171 613	243 754
4	36 585	72 833	169 849	231 745
8	36 585	73 369	160 696	242 317

Table 5. Number of times consistency was called for the constraints in 100-Sudoku when searching for 5 000 solutions.

Consistency Threads	Search Threads			
per Search Thread	1	2	4	8
1	364 718	435 524	1 102 162	1 613 385
2	721 723	933 104	2 453 025	1 604 395
4	720 976	925 494	2 089 093	1 571 044
8	720 980	920 276	1 731 205	1 470 914

Table 6. Number of search nodes explored when searching for 200 solutions to 100-Sudoku.

Consistency Threads	Search Threads			
per Search Thread	1	2	4	8
1	35 953	34 914	44 473	52 382
2	35 953	35 394	41 669	45 467
4	35 953	35 358	41 785	45 380
8	35 953	35 296	40 949	45 832

Table 7. Number of search nodes explored when searching for 5 000 solutions to 100-Sudoku.

Consistency Threads per Search Thread	Search Threads 1	2	4	8
1	763 827	784 204	958 969	1 002 032
2	763 827	784 980	920 223	881 475
4	763 827	785 547	915 305	894 489
8	763 827	784 443	923 470	886 828

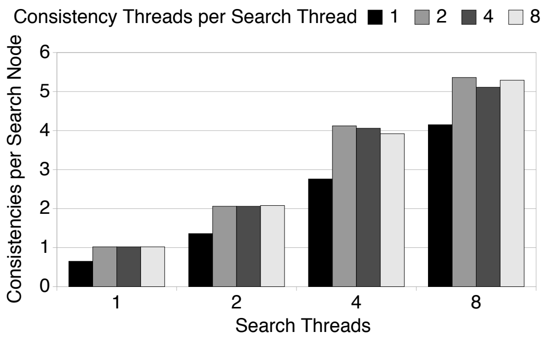

Fig. 9. Consistency enforcements per search node when searching for 200 solutions to Sudoku.

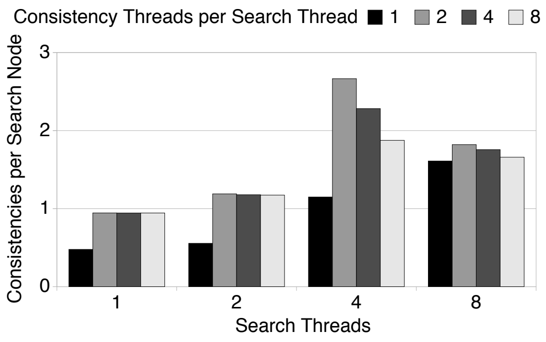

Fig. 10. Consistency enforcements per search node when searching for 5 000 solutions to Sudoku.

5.3 Results for Queens

It is much harder to achieve an even load-balance for Queens than for Sudoku. The structure of Queens is quite different from Sudoku. In Sudoku we only have global constraints with a high time complexity. In Queens, there are lots of small constraints to calculate the diagonals. Hence, for most of the execution, we have a very low processor load if we only use parallel consistency [11].

We used Queens in order to illustrate how parallel consistency can be useful when parallel search is not. Problems with little need for parallel consistency have more room for the parallel search threads to execute. However, Queens is a highly constrained problem. Even with 550 queens, there are very few search nodes that need to be explored. Hence, parallel search will usually only add overhead. However, adding parallel consistency can compensate for the performance loss.

As shown in Table 8 and Fig. 11, parallel search reduces performance. However, parallel consistency gives a speed-up even when we loose performance because of parallel search. We can also see that adding search threads can lead to sudden performance drops. This is largely because we end up overloading the memory bus and the processor cache. For eight search threads the performance increases compared to four threads. The reason is that we find a solution in a more easily explored part of the search tree.

Table 9, Table 10, and Fig. 12 all support our earlier observation that the workload increases heavily if we use barrier synchronization. The results come from that we have to evaluate the simple constraints many more times if we do not share data between them and the alldifferent constraints. The reason why we still get a speed-up is that the alldifferent constraints totally dominate the execution time and do not have to be run that much more often in parallel consistency.

Table 8. Execution times in seconds when searching for one solution to 550-Queens.

Consistency Threads	Search Threads			
per Search Thread	1	2	4	8
1	107	109	464	325
2	95	101	454	191
4	**77**	82	405	213
8	**77**	82	426	215

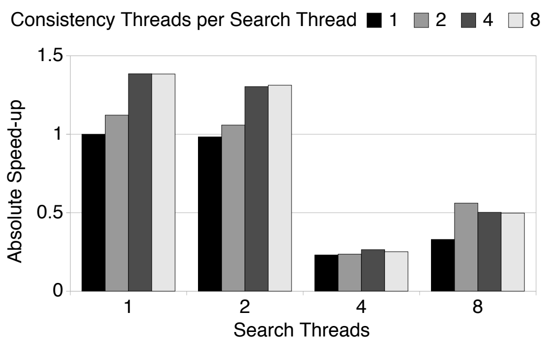

Fig. 11. Speed-up when searching for one solution to 550-Queens.

Table 9. Number of times consistency was called for the constraints in 550-Queens when searching for one solution.

Consistency Threads	Search Threads			
per Search Thread	1	2	4	8
1	322 415	662 392	2 475 709	2 560 781
2	772 585	1 566 891	5 551 542	6 159 671
4	771 537	1 554 595	5 182 159	6 153 881
8	769 972	1 543 778	5 014 605	6 152 789

Table 10. Number of search nodes explored when searching for one solution to 550-Queens.

Consistency Threads	Search Threads			
per Search Thread	1	2	4	8
1	1 246	2 624	20 866	11 200
2	1 246	2 787	23 114	10 193
4	1 246	2 735	22 072	10 292
8	1 246	2 834	23 025	10 591

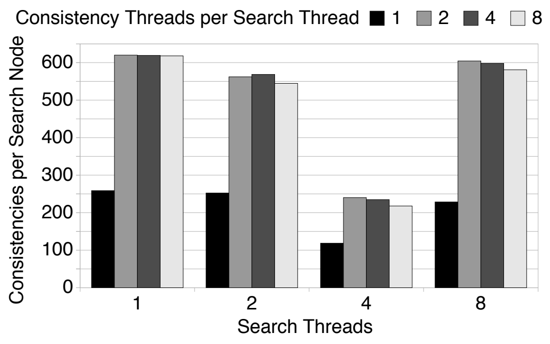

Fig. 12. Consistency enforcements per search node when searching for one solution to Queens.

6 Conclusions

The main conclusion is that it is possible to successfully combine parallel search and parallel consistency. However, it is very hard to do so. The properties of a problem, and size of the search space determines whether parallelism is useful or not. When trying to add two different types of parallelism, these factors become doubly important.

In general, if a problem is highly constrained, there is little room to add parallel search. If it is not constrained enough, there will be too many inconsistent branches for successfully adding parallel consistency. Finally, if a problem is reasonably constrained, the size of the search space, the uniformity of constraints, and the time complexity of the consistency algorithms determine whether fruitfully combining parallel search and parallel consistency is feasible.

In order to make sure that parallel consistency becomes less problem dependent, the need for synchronization must be reduced. This requires data to be shareable between global constraints during their execution. Since pruning is monotonic, this should be possible. However, it depends on the internal data structures used by the consistency algorithms. Hence, parallel consistency algorithms for each constraints may be a better direction of future research. Another interesting aspect is how much the order in which the constraints are evaluated matter to the performance. This is especially important for inconsistent states.

References

1. Marriott, K., Stuckey, P.J.: Introduction to Constraint Logic Programming. MIT Press, Cambridge, MA, USA (1998)
2. Dechter, R.: Constraint Processing. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2003)
3. Rolf, C.C., Kuchcinski, K.: Load-balancing methods for parallel and distributed constraint solving. Cluster Computing, 2008 IEEE International Conference on (2008) 304–309
4. Chu, G., Schulte, C., Stuckey, P.J.: Confidence-based work stealing in parallel constraint programming. In Gent, I., ed.: Fifteenth International Conference on Principles and Practice of Constraint Programming. Volume 5732 of Lecture Notes in Computer Science., Lisbon, Portugal, Springer-Verlag (2009) 226–241
5. Schulte, C.: Parallel search made simple. In Beldiceanu, N., Harvey, W., Henz, M., Laburthe, F., Monfroy, E., Müller, T., Perron, L., Schulte, C., eds.: Proceedings of TRICS: Techniques foR Implementing Constraint programming Systems, a post-conference workshop of CP 2000, Singapore (2000)
6. Michel, L., See, A., Hentenryck, P.V.: Parallelizing constraint programs transparently. In Bessiere, C., ed.: CP. Volume 4741 of Lecture Notes in Computer Science., Springer (2007) 514–528
7. Rao, V.N., Kumar, V.: Superlinear speedup in parallel state-space search. In: Proceedings of the Eighth Conference on Foundations of Software Technology and Theoretical Computer Science, London, UK, Springer-Verlag (1988) 161–174
8. Sun, X.H., Chen, Y.: Reevaluating amdahl's law in the multicore era. J. Parallel Distrib. Comput. **70**(2) (2010) 183–188
9. Nguyen, T., Deville, Y.: A distributed arc-consistency algorithm. Sci. Comput. Program. **30**(1-2) (1998) 227–250
10. Ruiz-Andino, A., Araujo, L., Sáenz, F., Ruz, J.J.: Parallel arc-consistency for functional constraints. In: Implementation Technology for Programming Languages based on Logic. (1998) 86–100
11. Rolf, C.C., Kuchcinski, K.: Parallel consistency in constraint programming. PDPTA '09: The 2009 International Conference on Parallel and Distributed Processing Techniques and Applications **2** (2009) 638–644
12. Schulte, C., Carlsson, M.: Finite domain constraint programming systems. In Rossi, F., van Beek, P., Walsh, T., eds.: Handbook of Constraint Programming. Foundations of Artificial Intelligence. Elsevier Science Publishers, Amsterdam, The Netherlands (2006) 495–526
13. Puget, J.F.: A fast algorithm for the bound consistency of alldiff constraints. In: AAAI/IAAI. (1998) 359–366
14. Kuchcinski, K.: Constraints-driven scheduling and resource assignment. ACM Transactions on Design Automation of Electronic Systems (TODAES) **8**(3) (2003) 355–383

Implementing Efficient Propagation Control

Christian Schulte[1] and Guido Tack[2]

[1] KTH – Royal Institute of Technology, Sweden
[2] Katholieke Universiteit Leuven, Belgium

Abstract. In propagation-based constraint solvers, *propagators* implement constraints by removing inconsistent values from variable domains. To make propagation efficient, modern constraint solvers employ two mechanisms of propagation control, *event-based* and *prioritized* propagation. Events, such as a bounds change of a particular variable, control which propagators need to be scheduled for re-evaluation. Prioritization controls which of the scheduled propagators is executed next.
While it has been shown that the combination of event-based and priority-based scheduling is an efficient approach for propagation control, this is the first publication on the implementation details of such a system. This paper presents the design of efficient data structures for propagator priority queues and the event system. The paper introduces the notions of *modification events* and *propagation conditions*, which refine the event-based model of propagation and yield an efficient implementation. The presented architecture is the basis of Gecode.

1 Introduction

In propagation-based constraint solvers, each constraint is implemented by a *propagator*, whose task it is to prune variable domains, removing values that are inconsistent with the implemented constraint. The constraint solver executes the propagators in turn until none of them can prune any domain any longer, thus establishing a *fixpoint*.

The fixpoint computation is the core of the solving process, its efficiency in terms of runtime and memory is vital for a solver's performance. Apart from actually executing the propagators, the solver has to perform efficient **propagation control**, which has two aspects: (1) if the propagator has modified some variable domains, the solver decides which other propagators it has to reconsider, or *schedule*, for propagation due to the changes; and (2) the solver picks one of the scheduled propagators for execution.

The standard technique for (1) is to specify which *events* a propagator *depends on.* An event describes how a particular variable has been changed, and a propagator that depends on a certain event will only be scheduled if that event happens. For example, a propagator for the constraint $x \leq y$ only depends on the events that the lower bound of x or the upper bound of y changes. In any other case, it cannot prune any variable domain and is thus at a fixpoint.

For (2), it is essential to organize the scheduled propagators in a *queue* (as opposed to a stack, which can suffer from starvation). Furthermore, it can be beneficial to *prioritize* propagators by their estimated runtime cost of propagation, first picking cheap propagators, and only later executing more expensive ones that can then take advantage of the cheap pruning done before.

Schulte and Stuckey [16] have performed a thorough evaluation of event-based and prioritized propagation.

Contributions. This paper develops implementation techniques and concrete data structures for efficient event-based, prioritized propagation control. It establishes the notions of *modification events* and *propagation conditions*, which are used to describe sets of events in a compact way. The paper develops indexed dependency arrays as an efficient data structure for the dependencies, and priority bucket queues for prioritized propagator scheduling. The developed architecture has been fully implemented and is the basis of Gecode [5].

Plan. After setting up the preliminaries in Section 2, Section 3 introduces propagation conditions and modification events, and Section 4 develops indexed dependency arrays. Section 5 discusses prioritized propagation and presents the priority bucket queue. Section 6 discusses related work, and Section 7 concludes the paper.

2 Preliminaries

This section recapitulates propagator-centered propagation, presenting the basic notions such as variables, propagators, and events.

Variables, constraints, domains, propagators. Constraint problems are modeled in terms of *variables*, representing the objects of the problem, and *constraints*, representing the relations that the objects are engaged in. In propagation-based constraint solvers, each variable has a *variable domain*, a set of values that it can take.

A *domain* d is a function mapping variables to sets of values. The set of values in d for a particular variable x, $d(x)$, is called the *variable domain* of x. A domain d is *stronger* than a domain d', written $d \subseteq d'$, iff for all variables x, $d(x) \subseteq d'(x)$.

Each constraint is implemented by a *propagator*, whose task it is to *prune* the domains. A propagator is a function that takes a domain and returns a stronger domain, possibly removing inconsistent values from the variable domains.

The constraint solver executes the propagators in turn until none of them can contribute any more pruning. The system thus reaches a mutual fixpoint. For a discussion of the properties of propagators that guarantee fixpoints, see [17]. Constraint propagation is incomplete, so the solver interleaves propagation with tree search, which yields a sound and complete solution procedure.

A propagator is *subsumed* in a domain d if it is at a fixpoint for any stronger domain $d' \subseteq d$, i.e., $p(d') = d'$. As subsumed propagators do not prune in the remaining subtree of the search, they can be removed.

In this paper, we will use the terms variable, domain, and propagator more freely, sometimes referring to the mathematical objects introduced above, sometimes referring to objects as implemented in a concrete constraint solver.

Events. An event describes how a variable domain was modified. Events are used to determine which propagators have to be re-executed when a variable domain is modified.

In the most basic event system, propagators only notify the solver of which variables they are *interested in*. For example, a propagator for the constraint $x \leq y$ only needs to be re-executed if either x or y has been modified since its last execution.

Most solvers use more complex events, which not only describe *which* domains have changed, but also *how* they have changed. A typical system for integer variables may consist of the events asn (the variable was assigned a value), lbc (the lower bound changed), ubc (the upper bound changed), and dmc (the domain changed). Events can overlap, for example, whenever the event lbc happens, also dmc happens. Looking again at the $x \leq y$ example, the propagator only needs re-execution when a lbc event on x or a ubc event on y happens.

An event e is characterized by a condition $e(d(x), d'(x))$ for two variable domains $d'(x) \subseteq d(x)$. For variable domains $d''(x) \subseteq d'(x) \subseteq d(x)$, it must satisfy $e(d(x), d''(x))$ if and only if $e(d(x), d'(x))$ or $e(d'(x), d''(x))$. An event always describes an actual domain modification: if $d(x) = d'(x)$, then $e(d(x), d'(x))$ must be false.

We define the set $\text{events}(d(x), d'(x)) := \{e \mid e(d(x), d'(x))\}$ for two variable domains $d'(x) \subseteq d(x)$. This construction ensures that events are *monotonic*, they are never discarded by further changes to a variable domain. For three variable domains $d''(x) \subseteq d'(x) \subseteq d(x)$, monotonicity implies that $\text{events}(d(x), d''(x)) = \text{events}(d(x), d'(x)) \cup \text{events}(d'(x), d''(x))$.

We say that an event e *implies* an event e' (written $e \rightarrow e'$) if and only if for all variable domains $d(x)$ and $d'(x)$, $e(d(x), d'(x))$ implies $e'(d(x), d'(x))$.

Our example event system features the following implications:

$$\begin{tikzcd}
 & \mathsf{dmc} & \\
\mathsf{lbc} \arrow[ur] & \mathsf{asn} \arrow[u] & \mathsf{ubc} \arrow[ul]
\end{tikzcd}$$

A simple propagation loop. The goal of the propagation loop is to compute a fixpoint of all propagators. Instead of re-executing all propagators in a loop, constraint solvers typically keep track of which propagators are known to be at a fixpoint; these are called *idle*. All other propagators are called *active* and are kept in a data structure called the *agenda*. The propagation loop picks a propagator p from the agenda, executes it, and determines which other propagators need to be put on the agenda again, based on what kind of events p caused. When the agenda is empty, propagation stops with a mutual fixpoint of all propagators.

The following pseudo-code is a high-level implementation of the propagation loop, assuming an object-oriented implementation language where q is an agenda of propagator objects, and each propagator object has a method `propagate` that (destructively) performs the propagation. This code leaves out the details of how the agenda q is implemented, or how the dependent propagators P are computed exactly. The rest of this paper fills in these details.

```
fixpoint()
1 while not q.empty()
2     p = q.head()
3     q.idle(p)
4     p.propagate()
5     P = propagators depending on events generated by p
6     ∀p′ ∈ P : q.enqueue(p′)
```

Recomputation versus trailing. Constraint solvers *backtrack* during search: The solver builds one path in the tree, and if it hits a failed state, it returns to a previous search state, chooses a different alternative, and continues the search.

Backtracking can be implemented using two different techniques, *trailing* and *recomputation* [15]. The former relies on storing *undo* information, which can be used to revert all the changes made during propagation and search between the failed state and the state that the solver is backtracking to. The latter, on the other hand, stores information how to *redo* the steps between a copy of the state higher up in the tree and the backtracking target state.

This paper describes the architecture of the Gecode [5] propagation kernel, which is based on copying and recomputation. The data structures are therefore designed to be memory efficient, because in a copying system, memory efficiency immediately yields a runtime advantage, too.

3 Modification Events and Propagation Conditions

This section introduces the notions of *modification events* and *propagation conditions*, which both capture different sets of events that a solver needs for propagator scheduling.

In order to perform propagator scheduling, the solver needs to record which events happen on which variables, and then determine the propagators that depend on these events. We are thus dealing with two different types of sets of events. A *modification event* is the set of events that happen when a variable domain is modified. A *propagation condition* is the set of events on a particular variable that a certain propagator depends on.

For example, when increasing the lower bound of an integer variable, the corresponding modification event may be $\{\mathsf{lbc}, \mathsf{dmc}\}$. A propagator that reacts to changes of either bound of a variable x has propagation condition $\{\mathsf{lbc}, \mathsf{ubc}\}$ on x. The solver schedules those propagators where for any variable, the intersection of the modification event and the propagation condition is not empty.

Propagator scheduling is one of the critical operations of a constraint solver kernel. The goal is therefore to make two operations as efficient as possible:

maintaining the set of events that has happened, and computing its intersection with the propagation conditions.

This can be achieved by representing each modification event and each propagation condition by an integer that can be used as an index into an appropriate data structure that represents the dependencies, as developed in the following section. The remainder of this section shows how to arrive at minimal definitions for propagation conditions and modification events, in order to keep the integer encoding as small as possible.

Propagation conditions. Some event sets are equivalent for the purpose of propagator scheduling. For example, the event sets $\{\mathsf{lbc}, \mathsf{dmc}\}$ and $\{\mathsf{dmc}\}$ are equivalent, as lbc implies dmc. More generally, if a propagator should be scheduled by an event e, then it will also be scheduled by any event e' such that $e' \to e$. We therefore restrict propagation conditions to the equivalence classes with respect to reverse implication:

A *propagation condition* π is a set of events that is closed under the converse of implication: for any two events e and e', if $e \in \pi$ and $e' \to e$, then $e' \in \pi$.

Our example event system yields the following propagation conditions:

$$\begin{array}{ccc}
 & \{\mathsf{asn}, \mathsf{lbc}, \mathsf{ubc}, \mathsf{dmc}\} & \\
 & \{\mathsf{asn}, \mathsf{lbc}, \mathsf{ubc}\} & \\
\{\mathsf{asn}\} & \{\mathsf{asn}, \mathsf{lbc}\} & \{\mathsf{asn}, \mathsf{ubc}\} \\
\{\mathsf{lbc}\} & \{\mathsf{ubc}\} & \{\mathsf{lbc}, \mathsf{ubc}\}
\end{array}$$

Modification events. Event-directed scheduling requires determining the set $\mathrm{events}(d(x), d'(x))$ when the domain changes from d to d'. Instead of computing this set from two given domains, an implementation will maintain a set of events incrementally during propagation. Such a set of events represents the modifications between two variable domains, and we call it a *modification event*.

Similar to propagation conditions, not all sets of events actually occur as modification events in practice. For instance, the set $\{\mathsf{lbc}\}$ cannot occur, as an lbc event always implies that a dmc event has happened, too. In general, we therefore define that a *modification event me* is a set of events that is closed under implication.

The definition implies that modification events are closed under union. This makes it easy to maintain the set of events incrementally for any variable. Again, as there are only few events in typical event systems, we can enumerate all modification events for a particular event system. An implementation can therefore represent modification events as small integers. In our example event system, we can simplify even further, as the asn event always implies *either* lbc or ubc. Therefore, the modification event $\{\mathsf{asn}, \mathsf{dmc}\}$ does not have to be represented.

The following modification events are therefore derived from the example event system:

$$\begin{array}{ccc}
 & \{\mathsf{asn}, \mathsf{lbc}, \mathsf{ubc}, \mathsf{dmc}\} & \\
\{\mathsf{asn}, \mathsf{lbc}, \mathsf{dmc}\} & \{\mathsf{lbc}, \mathsf{ubc}, \mathsf{dmc}\} & \{\mathsf{asn}, \mathsf{ubc}, \mathsf{dmc}\} \\
\{\mathsf{lbc}, \mathsf{dmc}\} & \{\mathsf{dmc}\} & \{\mathsf{ubc}, \mathsf{dmc}\}
\end{array}$$

The asn event. The event system presented above can be simplified further by treating the asn event as if it implies any other event:

$$\begin{tikzcd}
& \mathsf{dmc} & \\
\mathsf{lbc} \arrow[ur] & & \mathsf{ubc} \arrow[ul] \\
& \mathsf{asn} \arrow[uu] \arrow[ul] \arrow[ur] &
\end{tikzcd}$$

Consequently, the propagation conditions are now:

$$\begin{array}{ccc} & \{\mathsf{asn}, \mathsf{lbc}, \mathsf{ubc}, \mathsf{dmc}\} & \\ & \{\mathsf{asn}, \mathsf{lbc}, \mathsf{ubc}\} & \\ \{\mathsf{asn}\} & \{\mathsf{asn}, \mathsf{lbc}\} & \{\mathsf{asn}, \mathsf{ubc}\} \end{array}$$

And the modification events look as follows:

$$\begin{array}{ccc} & \{\mathsf{asn}, \mathsf{lbc}, \mathsf{ubc}, \mathsf{dmc}\} & \\ & \{\mathsf{lbc}, \mathsf{ubc}, \mathsf{dmc}\} & \\ \{\mathsf{lbc}, \mathsf{dmc}\} & \{\mathsf{dmc}\} & \{\mathsf{ubc}, \mathsf{dmc}\} \end{array}$$

In this simplified system, all propagators are executed at least once when all variables are assigned. In a solver that is based on copying, this additional strong invariant proves useful because it means that all propagators will eventually test for and report subsumption. Subsumed propagators are removed from the system and therefore do not have to be copied any longer, which saves memory and runtime. Gecode uses the simplified system for all its variable types[3].

4 Dependencies

This section develops the data structures that represent the propagator dependencies.

The propagation loop in Section 2 defined scheduling in a high-level way:

P = propagators depending on events generated by p
$\forall p' \in P: \; q.\texttt{enqueue}(p')$

Instead of collecting the events and then computing the set of propagators to schedule, we embed the scheduling into the variable modification operations. Whenever a propagator updates a variable domain in its `propagate` method, the variable computes the corresponding modification event *me* and schedules all propagators with corresponding propagation conditions.

The dependency data structure must therefore provide three basic operations:

– $x.\texttt{subscribe}(p, \pi)$ adds the propagator p to the dependencies of x at propagation condition π.

[3] For integer variables, Gecode uses a single event bnd instead of two separate events lbc and ubc. This was shown to be sufficient and increase efficiency [16]. We chose the more complicated setup for this paper to be able to discuss the full system.

- x.`cancel`(p, π) removes the propagator p from the dependencies of x at propagation condition π.
- x.`schedule`(π_i, π_j) iterates over the propagators between propagation conditions π_i and π_j of x to schedule them.

The most important operation is iteration. It is performed whenever an event happens, so we will design the data structure to be as efficient as possible in this case. For subscription and canceling, efficiency is not quite as important, as they happen less frequently.

We enforce a strong contract between propagators and the dependencies. A propagator must not cancel subscriptions that it has not established before, and it must cancel all its subscriptions when it ceases to exist (e.g. because it detects subsumption). We will discuss below why the invariants enforced by this contract are important.

Furthermore, the data structure must be *backtrackable*, i.e., the solver must be able to revert it to a previous state. We will discuss how to achieve this using either copying or trailing.

4.1 Indexed dependency arrays

The most efficient data structure for fast iteration is an array. So, in principle, we could have one array of propagators per propagation condition. However, in practice a single modification event often triggers several propagation conditions.

The dependencies are therefore stored in a single *dependency array* `dep`, sorted by propagation condition. In addition, we maintain the *dependency index* `idx`, which partitions the dependency array by propagation condition. Figure 1 shows this architecture.

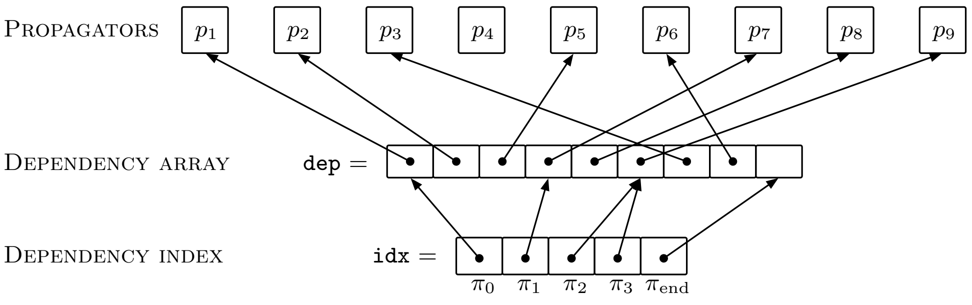

Fig. 1: Dependency data structures

For each propagation condition π_i, the dependency index points to the first propagator in the dependency array that is subscribed with π_i. For example, the first propagator subscribed with π_1 in Figure 1 is `dep`[`idx`[π_1]] $= p_7$. To iterate

```
subscribe($p, \pi_i$)
1  for $j = k$ downto $i$
2      dep[idx[$\pi_{j+1}$]] = dep[idx[$\pi_j$]]
3      idx[$\pi_{j+1}$] = idx[$\pi_{j+1}$] + 1
4  dep[idx[$\pi_i$]] = $p$
```

(a)

```
cancel($p, \pi_i$)
1  $j_p$ = idx[$\pi_i$]
2  while dep[$j_p$] $\neq p$ do $j_p = j_p + 1$
3  dep[$j_p$] = dep[idx[$\pi_{i+1}$] − 1]
4  for $j = i + 1$ to $k$
5      dep[idx[$j$] − 1] = dep[idx[$\pi_{j+1}$] − 1]
6      idx[$\pi_j$] = idx[$\pi_j$] − 1
7  idx[$\pi_{\mathrm{end}}$] = idx[$\pi_{\mathrm{end}}$] − 1
```

(b)

Fig. 2: Subscribing (a) and canceling (b)

over all propagators subscribed with a certain propagation condition π_i, we start at dep[idx[π_i]] and finish at dep[idx[π_{i+1}]-1]. There is one additional propagation condition, π_{end}, so that π_{i+1} and idx[π_{i+1}] are defined for all propagation conditions π_i.

Again for the example in Figure 1, scheduling all propagators that are subscribed with propagation condition π_1 would amount to scheduling the propagators p_7 and p_8. No propagator is subscribed with π_2 (as idx[π_2] $=$ idx[π_3]). Through this index data structure, iterating over all propagators subscribed with a particular propagation condition is as efficient as possible, taking constant time per propagator, and with low constants in practice.

We can now define the method schedule(π_i, π_j), which schedules all propagators starting at propagation condition π_i and finishing at propagation condition π_j. For the above example, schedule(π_0, π_2) would thus schedule p_1, p_2, p_5, p_7, and p_8. The following code implements schedule, assuming a method enqueue that puts a propagator into the right queue:

```
schedule($\pi_i, \pi_j$)
for $k$ = idx[$\pi_i$] to idx[$\pi_{j+1}$] − 1
    enqueue(dep[$k$])
```

4.2 Subscribing and cancelling

The remaining operations to be defined are subscribing and canceling. Subscribing a propagator p with propagation condition π_i means adding it at the appropriate position to the dependency array and modifying the index accordingly. Assuming that the dependency array is resized dynamically, subscription can be implemented to have amortized run-time $O(k-i)$ as shown in Figure 2(a). First, some space is cleared for the new subscription at dep[idx[π_i]] (lines 1–3). Then the new subscription is entered (line 4).

A subscription can be canceled in $O($idx[π_{i+1}] $-$ idx[π_i] $+ i)$ (Figure 2(b)). The **while** loop in line 2 finds the index of p in the dependency array (note that the loop is only correct if the propagator is actually subscribed to the variable). After finding the index j_p, the position $\texttt{dep}[j_p]$ is reused (lines 3–7).

Assigned variables. Variables that are assigned, i.e., whose domain is a singleton, cannot produce any events any more. Therefore, subscribing and canceling on these variables is useless, and the overhead can be avoided by a simple check.

Scheduling upon subscription. A propagator typically subscribes to its variables when it is created. A newly created propagator must be scheduled for execution (otherwise, no propagation would happen for the first fixpoint). There is one exception: If the propagator subscribes only with $\{\mathsf{asn}\}$ propagation conditions and none of the variables is assigned, it does not have to be scheduled.

Rewriting. A special case of canceling and subscribing is *propagator rewriting*. E.g., consider the case when all but two variables in a long linear equation are assigned. Then the propagator for the linear equation can be rewritten to a simpler, more efficient binary version. The old propagator cancels its subscriptions, and the new propagator subscribes. Note that this order is essential: it means that the dependency arrays will not need resizing, as the old propagator leaves enough space for the new one.

4.3 Scheduling

When a variable domain is modified resulting in a modification event me, the variable determines which ranges of propagation conditions intersect with me (this is implemented as a lookup table), and then schedules each of those ranges π_i, π_j using $\texttt{schedule}(\pi_i, \pi_j)$.

Assume the following enumeration of the propagation conditions of our simplified event system:

$$\pi_4 = \{\mathsf{asn}, \mathsf{lbc}, \mathsf{ubc}, \mathsf{dmc}\}$$
$$\pi_3 = \{\mathsf{asn}, \mathsf{lbc}, \mathsf{ubc}\}$$
$$\pi_0 = \{\mathsf{asn}\} \qquad \pi_1 = \{\mathsf{asn}, \mathsf{lbc}\} \qquad \pi_2 = \{\mathsf{asn}, \mathsf{ubc}\}$$

The scheduling then looks as follows:

```
case $me$ of
    $\{\mathsf{asn}, \mathsf{lbc}, \mathsf{ubc}, \mathsf{dmc}\}$: $x.\texttt{schedule}(\pi_0, \pi_4)$
    $\{\mathsf{lbc}, \mathsf{ubc}, \mathsf{dmc}\}$: $x.\texttt{schedule}(\pi_1, \pi_4)$
    $\{\mathsf{lbc}, \mathsf{dmc}\}$: $x.\texttt{schedule}(\pi_1, \pi_1)$; $x.\texttt{schedule}(\pi_3, \pi_4)$
    $\{\mathsf{ubc}, \mathsf{dmc}\}$: $x.\texttt{schedule}(\pi_2, \pi_4)$
    $\{\mathsf{dmc}\}$: $x.\texttt{schedule}(\pi_4, \pi_4)$
```

4.4 Copying and trailing

Dependencies can be modified dynamically. For example, a propagator may "lose interest" in some of its variables, if it can determine that no further change of their domains will cause any propagation. Or, propagators can replace themselves with simpler versions. Or, in the most dynamic case, the propagator only needs subscriptions to a dynamically changing subset of the variables, such as for the *watched literals* technique [12,7].

In most of these cases, the dependencies must be *backtrackable*. We can achieve this either by *copying* them, or by *trailing* any changes [15].

Copying. When copying the dependency arrays, it is advantageous to allocate enough memory for all dependency arrays of all variables in one block. That way, the copy will be compact and the overhead for allocation is low.

Trailing. Dynamic dependencies change infrequently (typically much less than variable domains). A simple trailing scheme that stores a function pointer together with the data which dependency needs to be changed works well [15].

5 Propagator Priority Queue

This section develops a priority queue data structure that provides efficient operations for priority-based propagator scheduling. We first recapitulate priority-based propagator scheduling, and then present the priority bucket queue.

5.1 Propagator priorities

Schulte and Stuckey showed [16] that propagator priorities are an important technique for efficient propagator scheduling[4].

Obviously, prioritized propagation requires some measure to determine a propagator's priority. For this paper, we assume that priorities model the estimated *runtime cost* of propagation. The execution of propagators with high estimated runtime cost is postponed, so that they can take advantage of the pruning of the cheaper propagators that are run first.

A straightforward way to estimate the cost is to classify the propagators according to their algorithmic complexity. We will use the following system of costs and priorities: `unary` = 7, `binary` = 6, `ternary` = 5, `linear` = 4, `quadratic` = 3, `cubic` = 2, `veryslow` = 1. The names suggest the arity of the corresponding propagator (for the highest three priorities), or the asymptotic run-time for n-ary propagation algorithms. The cost of propagation often changes dynamically. For instance, a typical algorithm for propagating linear equations has an asymptotic run-time linear in the number of *unassigned* variables. Accordingly, when all but three variables are assigned, the cost should be reported as `ternary` instead of `linear`.

[4] Priorities are particularly useful for implementing *staged propagation* [16], which is out of the scope of this paper, but can be implemented easily on top of the presented prioritized, event-based system.

5.2 Priority bucket queue

For our purposes, the priority queue must provide four operations:

– `enqueue`(p) adds propagator p at the priority determined by its `cost` method. If p is already in the queue, it is re-prioritized according to its current cost.
– `empty`() tests whether the queue is empty.
– `head`() returns the oldest propagator at the highest priority.
– `idle`(p) removes propagator p from its current queue and marks it as idle.

All four operations are performed extremely often during the fixed point computation, and are hence crucial for the solver's performance.

Common algorithms for priority queues are based on variations of the *heap* data structure (see for example [11,3]). Heaps support an arbitrary number of priority levels. For most types of heaps, the run-time complexity of the enqueue and dequeue operations depends on the number of elements in the queue. For example, using binary heaps, both operations require time in $O(\log n)$ if n is the number of elements in the queue.

In order to make `enqueue` and `dequeue` as efficient as possible, we restrict the number of priority levels to a small, fixed set of integers (such as the cost values introduced above). Then, a priority queue based on *buckets* can be used (see [11]), providing constant-time `enqueue` and `dequeue` operations. We will now see how a bucket-based priority queue of propagators can be implemented.

The bucket queue. A bucket-based priority queue consists of an array of doubly-linked lists of propagators. The list at array index i represents the queue of propagators at priority i. Furthermore, a propagator can only be in one queue at a time. We can hence embed the links for the doubly-linked lists into the propagator objects. In addition to the lists for each priority, the solver maintains the list of idle propagators, the so-called *idle queue* (in our case modeled as priority 0). The invariant is then that a propagator is always in exactly one queue.

Each list of propagators is cyclic and terminated by a *sentinel element*. The sentinels are kept in an array that represents the priority queue. Figure 3 depicts an example of this architecture. An empty queue is depicted as a sentinel with a simple cycle (as at priority $k-1$).

This implementation of a bucket queue yields efficient access. Inserting and removing a propagator can be done in constant time—unlink it from its current queue, and link it at the position before the sentinel element of the target queue. Finding the next propagator to schedule costs at most k tests. Queues are managed as follows:

– The solver can access the queue with priority i as $Q[i]$.
– p.`next`() returns the propagator following p in the linked list.
– p.`unlink`() removes propagator p from its current queue.
– $Q[i]$.`tail`(p) adds p as the last propagator to the queue with priority i.

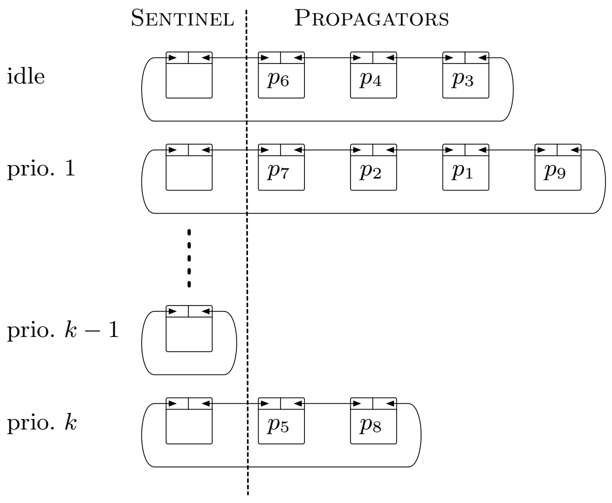

Fig. 3: Propagators in prioritized queues

A propagator is added to the queue that corresponds to its cost, which it reports using the `cost` method. The following code implements the `enqueue`, `head`, and `idle` methods, as well as a method `empty` that reports whether all queues except the idle queue are empty, indicating that propagation has reached a fixpoint.

```
enqueue($p$)
1  $p$.unlink()                 // remove $p$ from current queue
2  $Q[p$.cost()].tail($p$)       // put $p$ into new queue
```

```
head()
1  for $i = k$ downto 1
2      if $Q[i]$.next() $\neq Q[i]$ then return $Q[i]$.next()
```

```
idle($p$)
1  $p$.unlink()                 // remove $p$ from current queue
2  $Q[0]$.tail($p$)              // put $p$ into idle queue
```

```
empty()
1  for $i = k$ downto 1
2      if $Q[i]$.next() $\neq Q[i]$ then return FALSE
3  return TRUE
```

Avoiding re-scheduling of propagators. The code above schedules a propagator anytime its propagation condition matches a modification event. This involves unlinking the propagator, re-evaluating its cost, and putting it into the queue again. If the propagator already was in the queue due to a previous variable modification, the overhead of re-scheduling it should be avoided.

As an indication of whether the propagator is already in the correct queue, we will use the set of events since its last invocation, called the *modification event delta* Δme, and store it in every propagator. Before executing a propagator, it is set to the empty set. The `schedule` method then takes the modification event that caused the scheduling as an additional argument:

```
schedule($\pi_i, \pi_j, me$)
1  for $k = \texttt{idx}[\pi_i]$ to $\texttt{idx}[\pi_{j+1}] - 1$
2     if $me \not\subseteq \texttt{dep}[k].\Delta me$ then
3        $\texttt{dep}[k].\Delta me = \texttt{dep}[k].\Delta me \cup me$
4        enqueue(dep[$k$])
```

Line 2 makes sure that a propagator is only added to the queue if the new modification event *me* is not already contained in the propagator's modification event delta. This is correct because if $me \subseteq \texttt{dep}[k].\Delta me$, then the propagator has already been put into the queue before through line 3.

Memory and run-time efficiency. The bucket queue is as efficient as possible, both in terms of memory requirements and run-time. The asymptotic run-time for all operations is a small constant if we restrict the priorities to a small, fixed set. Priority-based scheduling with a fixed number of priorities is the standard in all propagation-based solvers (see Section 6), and has proven effective in practice. In terms of memory, this architecture requires two pointers per propagator for the doubly-linked list, which, theoretically, is an overhead of one pointer per propagator compared to an array-based implementation. However, using arrays for the queues would require dynamic resizing, which again costs memory and/or runtime. In practice, embedding the double links in the propagator objects is therefore without overhead.

6 Related Work

Most constraint solvers are based on propagator-centered, event-directed, prioritized propagation as presented in this paper.

▶ **SICStus Prolog** [1] employs a priority queue of propagators, using two priority levels.
▶ **Mozart OZ** [13] maintains a two-level priority queue of propagators, and scheduling is based on events. Mozart offers additional priorities for non-monotonic propagators (as described in [14]): each non-monotonic propagator gets its own priority level, effectively fixing the order in which non-monotonic propagators are run and hence maintaining the guarantee to compute a unique fixed point.
▶ **Eclipse Prolog** [18] has a feature called *suspension*, which attaches a Prolog goal to finite domain variables. When the variable domain changes, the goal, which may implement a propagator, is scheduled. The Eclipse system features twelve priority levels, but like SICStus and Mozart, its finite domain solver only makes use of two levels.

▶ **B-Prolog** [19] queues *action rules*, which correspond to propagator invocations. A particularity of B-Prolog is that the same propagator can appear several times in the queue, once for each variable that triggered its scheduling.
▶ **Choco** [9] provides a sophisticated priority system with seven levels and both FIFO and LIFO scheduling, but is not propagator-centered, as explained below.

Variable-centered propagation. In our setup, the agenda holds the propagators that are not necessarily at a fixed point. Some solvers, notably ILOG Solver [8], Choco [2], and Minion [6], use an alternative approach: an *agenda of modified variables* instead of an agenda of propagators. A solver that bases scheduling on an agenda of variables performs *variable-centered propagation.*

ILOG Solver, Choco, and Minion actually implement a hybrid approach. When a modified variable is taken from the queue, its dependent propagators can either be run immediately, or put into a queue of propagators.

The advantage of variable-centered over propagator-centered propagation is that whenever a propagator is invoked, the information which variable exactly triggered the propagation is directly available. The propagator can take this information into account in order to compute the new domain *incrementally*, without recomputing from scratch. Lagerkvist and Schulte [10] show how *advisors* can be used to implement incremental propagation in a propagator-centered system. Their implementation is a straightforward extension of the data structures presented in this paper.

Propagator queues. It is folklore knowledge that propagators should be scheduled in a FIFO fashion. Similarly, using events to prevent gratuitous scheduling of propagators has been used in constraint solvers for a long time—one can argue that it was already present in the early DPLL algorithm [4]. Schulte and Stuckey [16] perform detailed experiments with different agenda strategies as well as priority queues, substantiating this folklore knowledge with empirical evidence. They also provide a comprehensive study of events, including a detailed experimental evaluation of different event schemes, fixed point reasoning, and staged propagation.

7 Conclusions

This paper developed an architecture and concrete data structures for event-based, prioritized propagator scheduling. It introduced the notions of *modification events* and *propagation conditions*, which capture exactly the sets of events that occur during propagator scheduling. Based on these notions, the paper developed *indexed dependency arrays*, an efficient data structure for storing and accessing the dependency information.

Furthermore, the paper presents the design of *priority bucket queues*, which are used to implement propagator scheduling prioritized by estimated cost of propagation.

The presented data structures are the core of the Gecode constraint solver, one of the most efficient solvers available today.

Acknowledgements. The authors would like to thank Mikael Z. Lagerkvist for many discussions about details of the Gecode architecture, and the anonymous reviewers for helpful comments that improved this version of the paper.

References

1. Mats Carlsson, Greger Ottosson, and Björn Carlson. An open-ended finite domain constraint solver. In Hugh Glaser, Pieter H. Hartel, and Herbert Kuchen, editors, *PLILP'97*, volume 1292 of *LNCS*, pages 191–206. Springer, 1997.
2. CHOCO, 2010. http://choco-solver.net.
3. Thomas M. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 2nd ed. edition, 2001.
4. Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.
5. Gecode, generic constraint development environment, 2010. http://www.gecode.org.
6. Ian P. Gent, Christopher Jefferson, and Ian Miguel. Minion: A fast scalable constraint solver. In Gerhard Brewka, Silvia Coradeschi, Anna Perini, and Paolo Traverso, editors, *ECAI 2006*, pages 98–102. IOS Press, 2006.
7. Ian P. Gent, Christopher Jefferson, and Ian Miguel. Watched literals for constraint propagation in Minion. In Frédéric Benhamou, editor, *CP 2006*, volume 4204 of *LNCS*, pages 182–197. Springer, 2006.
8. ILOG Solver, part of ILOG CP, 2009. http://www.ilog.com/products/cp.
9. F. Laburthe. Choco: Implementing a CP kernel. In *TRICS*, pages 71–85, September 2000.
10. Mikael Z. Lagerkvist and Christian Schulte. Advisors for incremental propagation. In Christian Bessière, editor, *CP 2007*, volume 4741 of *LNCS*, pages 409–422. Springer, 2007.
11. Kurt Mehlhorn and Stefan Näher. *LEDA - A platform for combinatorial and geometric computing*. Cambridge University Press, 1999.
12. Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: engineering an efficient SAT solver. In *DAC '01*, pages 530–535, New York, NY, USA, 2001. ACM Press.
13. The Mozart programming system, 2009. http://www.mozart-oz.org.
14. Tobias Müller. *Constraint Propagation in Mozart*. Doctoral dissertation, Universität des Saarlandes, Naturwissenschaftlich-Technische Fakultät I, Fachrichtung Informatik, Saarbrücken, Germany, 2001.
15. Raphael M. Reischuk, Christian Schulte, Peter J. Stuckey, and Guido Tack. Maintaining state in propagation solvers. In Ian Gent, editor, *CP 2009*, volume 5732 of *LNCS*, pages 692–706. Springer, 2009.
16. Christian Schulte and Peter J. Stuckey. Efficient constraint propagation engines. *Transactions on Programming Languages and Systems*, 31(1):2:1–2:43, dec 2008.
17. Christian Schulte and Guido Tack. Weakly monotonic propagators. In Ian Gent, editor, *Proceedings of the 15th international conference on principles and practice of constraint programming*, volume 5732 of *LNCS*, pages 723–730. Springer, 2009.
18. Mark Wallace, Stefano Novello, and Joachim Schimpf. Eclipse: A platform for constraint logic programming. Technical report, IC Parc, Imperial College, London, 1997.
19. Neng-Fa Zhou. Programming finite-domain constraint propagators in action rules. *Theory and Practice of Logic Programming*, 6:483–507, 2006.

Handling Heterogeneous Constraints in Revision Ordering Heuristics

Julien Vion and Sylvain Piechowiak

Université de Valenciennes et du Hainaut Cambrésis,
LAMIH CNRS FRE 3304,
59313 Valenciennes Cedex 9, France.
`{julien.vion|sylvain.piechowiak}@univ-valenciennes.fr`

Abstract. Most constraint solvers use the general AC-5 scheme [17] to handle constraint propagation. AC-5 generalizes the concept of *constraint revision*. Each constraint type can thus be shipped with its own revision algorithm, with various complexities and performances.
Previous papers showed that the order in which constraints are revised have a non-negligible impact on performances of propagation [20,6,1]. However, most of the ideas presented on these papers are based on the use of homogeneous propagators for binary constraints defined in extension. This paper give ideas to handle heterogeneous constraints in a general revision schedule.

1 Introduction

The constraint satisfaction problem (CSP) consists in deciding whether a solution to a discrete constraint network (CN) exists. A CN consists in a set of discrete variables and constraints. Each constraint has one or more variables in its scope, and defines which instantiations of these variables are allowed. A solution to the CN is an instantiation of all variables which satisfies all constraints. The CSP is a standard NP-complete problem and generalizes very naturally many real-life industrial problems such as scheduling, rostering, etc.

Standard techniques for solving CSP instances use interleaved *decision*, *propagation* and *backtrack* steps. Most often, the propagation phase consists in establishing arc consistency by pruning all values that are inconsistent (i.e. cannot appear in any solution of the CSP according to the current decisions) from the point of view of a single given constraint. The resulting search algorithm is called `MAC` (see Page 2). The `AC` function performs the propagation step (see below), and returns **false** if the CN is inconsistent (e.g., one variable domain is empty). $\mathcal{N} = \top$ means that the CN is trivially consistent (e.g., there are no constraints, or all variable domains are singletons). In the given algorithm, δ should not be trivially implied by $\mathcal{N}$, and the final disjunction has a short-circuit behavior. This algorithm is of course very schematic. In particular, users are often interested in obtaining a solution, not simply knowing that one exists. Obtaining a consistent solution from the given algorithm is trivial. The nature of the decisions δ is a major issue in Constraint Programming. Standard all-purpose algorithms

Function MAC($\mathcal{N}$): boolean

```
1 $\mathcal{N}' \leftarrow \mathcal{N}$ ;
2 if $\neg$AC($\mathcal{N}'$) then return $false$;
3 if $\mathcal{N}' = \top$ then return $true$;
4 Let $\delta$ be some logical decision;
5 return MAC($\mathcal{N}'|\delta$) $\vee$ MAC($\mathcal{N}'|\neg\delta$);
```

usually make variable assignements (reduce the domain of a chosen variable to a singleton). The choice of the variable is important. Common decision heuristics are the *dom/ddeg* or *dom/wdeg* variable assignment ordering heuristics [5].

Removing values that are inconsistent w.r.t. a given constraint is called a *constraint revision* and propagation algorithms are designed to perform such revisions until some fix-point is reached. As constraint revisions are NP-hard in the general case (i.e. without any clue on the semantics of the constraint), for a long time, CSP were limited so as only to involve binary constraints (two variables per constraint). The binary CSP is NP-complete, but AC can be enforced on any binary CN in polynomial space and time.

Dozens of algorithms have been proposed during the last 40 years for performing the propagation step: AC-1 to AC-8, numerous variants of AC-3, etc. Most of these algorithms are actually the combination of a *propagation* and a general *constraint revision* algorithms (either NP-hard or limited to binary constraints). However, two propagation algorithms, AC-5 [17] and GAC-schema [3], consider a generic revision process, and thus generalize most other algorithms. In particular, AC-5 opened enormous perspectives to constraint programming. The main idea of AC-5 is that each constraint in the CN has semantic properties, that can be exploited by a specific (often polynomial) algorithm to perform the constraint revision. Thus, CSP solvers provide a "toolbox" of known useful constraints (*less than*, *not equal*, *sum*, *all different*...), each of which is shipped with its own *propagator*. This scheme gave birth to *global constraints* [2], which permitted to reuse powerful graph theory, artificial intelligence or operational research algorithms for performing constraint revision, and are now of primary importance to handle industrial-class problems. Most modern constraint solvers are based on AC-5.

AC-5, as all AC algorithms since AC-3 [12], are based on a *propagation queue*. When a variable loses a value, constraints involving this variable may no longer be AC, and requires to propagate the removed value to other variables. The propagation queue is used to keep track of such modifications, from which the propagation algorithms deduce which constraint revisions must be performed. The nature of the data stored in the queue (values, constraints and/or variables), as well as the order in which the different constraint revisions are processed, have a significant impact on the performance of the propagation. Works have been dedicated to devise a "good" ordering of the revisions [20,6,1], but they are usually focused on binary CSP with general propagators. Applying these techniques to heterogeneous propagators may lead to trivial pathological cases.

Note: in this paper, we distinguish *generic* from *general* concepts. A *generic* scheme can be *specialized* to the most efficient technique for the sought problem. A *general* algorithm works on any problem without specialization.

The contributions of this paper are :

1. establish a clear state-of-the-art on coarse-grained, generic propagation algorithms (Section 3),
2. define a generic, constraint-based revision ordering heuristic (Section 4.2),
3. survey data structures proposed in the algorithmic literature and show experimentally how they can improve the performance of propagation algorithms (Section 4.3).

2 Background

Definition 1 (Constraint Network, Variable, Domain, Constraint, Instantiation). *A* Constraint Network $\mathscr{N}$ *is a pair* $(\mathscr{X}, \mathscr{C})$ *which consists of :*

- *a set of* n variables $\mathscr{X}$*; a* domain $\mathrm{dom}(X)$ *is attached to each variable* $X \in \mathscr{X}$ *and denotes the finite set of at most* d *values that the variable* X *can be instantiated to, and*
- *a set of* e constraints $\mathscr{C}$*; each constraint* $C \in \mathscr{C}$ *involves at most* k *variables* $\mathrm{vars}(C) \subseteq \mathscr{X}$*; the constraint specifies the allowed* instantiations *for these variables.*

The set of constraints with a given variable X in scope is denoted $\mathrm{ctr}(X)$.

A constraint can be defined in *extension* (i.e., an exhaustive list of allowed or forbidden instantiations), or in *intention* (i.e., using some application $f_C : \prod_{X \in \mathrm{vars}(C)} \mathrm{dom}(X) \to \mathbb{B}$). *Global constraints* define some property of arbitrary arity that the values of the variables in its scope must verify (e.g., *all different*).

The *Constraint Satisfaction Problem* (CSP) consists in deciding whether a solution to a CN (i.e., an instantiation of all variables satisfying all constraints of the CN) exists. A *constraint check* consists in testing whether a constraint allows a given instantiation of variables. When all constraints can be checked in polynomial space and time, the CSP is NP-complete.

Definition 2 (Arc consistency). *Let* C *be a constraint and* $X \in \mathrm{vars}(C)$*. Value* $v \in \mathrm{dom}(X)$ *is* Arc-Consistent *(AC) w.r.t.* C *iff there exists an instantiation of* $\mathrm{vars}(C)$*, allowed by* C*, which instantiates* X *to* v *(such an instantiation is called a* support *of* v *w.r.t.* C*).* C *is AC iff* $\forall X \in \mathrm{vars}(C)$*,* $\forall v \in \mathrm{dom}(X)$*,* v *is AC.*

$\mathscr{N} = (\mathscr{X}, \mathscr{C})$ *is AC iff* $\forall C \in \mathscr{C}$*,* C *is AC.*

In the literature, the definition of Arc Consistency is often restricted to binary CNs, and the extension of AC to non-binary CNs is called Generalized AC, Hyper-AC, or Domain Consistency. In this paper we refer to Arc Consistency for both binary and non-binary CNs.

Definition 3 (Closure). *Let $\mathcal{N} = (\mathscr{X}, \mathscr{C})$ be a constraint network.* $\mathrm{AC}(\mathcal{N}, C)$ *is the* closure *of $\mathcal{N}$ for AC on C, i.e. the CN obtained from $\mathcal{N}$ where $\forall X \in$* $\mathrm{vars}(C)$*, all values $v \in \mathrm{dom}(X)$ that are not AC w.r.t. C have been removed.*

$\mathrm{AC}(\mathcal{N})$ *is the closure of $\mathcal{N}$ for AC, i.e. the CN obtained from $\mathcal{N}$ where $\forall C \in \mathscr{C}$, C have been made AC by closure.*

For any CN $\mathcal{N}(\mathscr{X}, \mathscr{C})$, $\mathrm{AC}(\mathcal{N}, C)$ for any $C \in \mathscr{C}$ and $\mathrm{AC}(\mathcal{N})$ are unique. In the general case, computing the closure for AC on a CN is NP-hard. Optimal algorithms such as GAC-schema are in $O(ekd^k)$ [3].

Definition 4 (Propagator). *Given a CN $\mathcal{N} = (\mathscr{X}, \mathscr{C})$, the* propagator *for a given constraint $C \in \mathscr{C}$ is the algorithm that computes* $\mathrm{AC}(\mathcal{N}, C)$*.*

3 A generic, coarse-grained propagation algorithm

This section does not intend to bring out innovative propagation algorithms, but instead aims to establish a clear state-of-the-art of coarse-grained, generic propagation techniques.

The main difference between AC algorithms lies in the way the general constraint propagator works. However, independently of the general propagator, these algorithms are often sorted in two families, depending on the nature of the data stored into the propagation queue. So-called *fine-grained* algorithms store every single *value* that have been removed from the domain of the different variables in the propagation queue, and try to exploit this information to avoid unecessary work. *Coarse-grained* algorithms only store the *variable* where a value have been removed, and/or constraints involving them, that thus must be revised. Although using fine-grained propagation queues is essential in designing optimal algorithms, the theoretical difference is at best marginal (the coarse-grained GAC-2001 algorithm is in $O(ek^2d^k)$ [4]), and the simpler data structures used by coarse-grained algorithms usually make them as much efficient in practice.

Mackworth's original AC-3 algorithm [12] was *arc-oriented*, that is, the propagation queue was composed of (*Variable*, *Constraint*) pairs. The *Variable* part of the pair identifies a variable which is not guaranteed to be AC w.r.t. the *Constraint*. McGregor showed in [13] that a similar behavior could be obtained by simply storing the *modified variables* in the queue. However, when working with non-binary constraints, variable-oriented propagation is not informative enough to avoid all useless revisions: when two variables involving the same non-binary constraint are in the queue, the domain of each variable involved by the constraint should be controlled for arc-consistency only once.

Boussemart *et al.* proposed in [6] to introduce an auxiliary data structure we call *modified*[C] to emulate the benefits of an arc-oriented propagation scheme in a variable-oriented propagation algorithm. It is used to keep track of which variables have been actually modified since the last revision of a constraint. Interestingly enough, this auxiliary data structure can also be used to devise a purely constraint-oriented propagation algorithm which avoids these useless

Algorithm 1: AC-5^v($\mathcal{N} = (\mathscr{X}, \mathscr{C})$) : CN

```
1  $Q \leftarrow \mathscr{X}$;
2  foreach $C \in \mathscr{C}$ do  $modified[C] \leftarrow \text{vars}(C)$;
3  while $Q \neq \emptyset$ do
4      Pick $X$ from $Q$;
5      foreach $C \in \text{ctr}(X)$ s.t. $modified[C] \neq \emptyset$ do
6          $\Delta \leftarrow C$.revise($modified[C]$) ;
7          if $\Delta = \bot$ then return false ;
8          $Q \leftarrow Q \cup \Delta$;
9          $modified[C] \leftarrow \emptyset$;
10         foreach $Y \in \Delta$ do
11             foreach $C' \in \text{ctr}(Y) \backslash C$ do
12                 $modified[C'] \leftarrow modified[C'] \cup \{Y\}$;
13 return true;
```

revisions. The version presented here is slightly optimized (with $O(k)$ overhead in Algorithm 3 against $O(k^2)$ in the original version).

The original AC-5 algorithm was fine-grained, so we propose our coarse-grained variants.

3.1 Variable-oriented propagation

In AC-5^v (Algorithm 1), the propagation queue Q contains recently modified variables, which require the revision of the constraints involving them (loop starting on Line 5). Initially, all variables are put in Q, however, when using the `MAC` procedure, only variables involved by the decisions δ are concerned.

The call to C.`revise`($modified[C]$) on Line 6 calls C's propagator, which may remove values from vars(C). The propagator returns a set $\Delta \subseteq \text{vars}(C)$ of modified variables,[1] or $\bot$ if an inconsistency has been detected (e.g., the domain of a variable has been emptied).

3.2 Constraint-oriented propagation

In this variant, called AC-5^c (Algorithm 2), constraints yet to be revised are stored in the queue. This leads to a somewhat simpler algorithm and finer queue, but the *modified* data structure is even more important to avoid unecessary work: when a constraint C is put in the queue due to some removals in the domain of a variable X involved by C, C's propagator only needs to control the domains of the *other* variables for arc consistency.

Algorithm 2: AC-5^{c}($\mathcal{N} = (\mathscr{X}, \mathscr{C})$) : CN

```
1  $Q \leftarrow \mathscr{C}$;
2  foreach $C \in \mathscr{C}$ do  $modified[C] \leftarrow \text{vars}(C)$;
3  while $Q \neq \emptyset$ do
4      Pick $C$ from $Q$;
5      $\Delta \leftarrow C.\texttt{revise}(modified[C])$ ;
6      if $\Delta = \bot$ then  return false ;
7      $modified[C] \leftarrow \emptyset$;
8      foreach $Y \in \Delta$ do
9          foreach $C' \in \text{ctr}(Y) \backslash C$ do
10             $Q \leftarrow Q \cup \{C'\}$;
11             $modified[C'] \leftarrow modified[C'] \cup \{Y\}$;
12 return true;
```

Algorithm 3: revise^{rm}($modified$: {Variable}): {Variable}

```
1  $\Delta \leftarrow \emptyset$;
2  foreach $X \in \text{vars}(this)$ s.t. $modified \neq \{X\}$ do
3      foreach $v \in \text{dom}(X)$ s.t. $this.res[X][v]$ is not valid do
4          $\tau \leftarrow this.\texttt{findSupport}(X, v)$ ;
5          if $\tau = \bot$ then
6              remove $v$ from $\text{dom}(X)$;
7              if $\text{dom}(X) = \emptyset$ then return $\bot$;
8              $\Delta \leftarrow \Delta \cup \{X\}$;
9          else
10             foreach $Y \in \text{vars}(this)$ do  $this.res[Y][\tau[Y]] \leftarrow \tau$;
11 return $\Delta$;
```

3.3 The AC-3$^{\text{rm}}$ propagator

To illustrate the use of our AC-5^{v}/AC-5^{c} scheme, we give a sample general propagator, called `revise`$^{\texttt{rm}}$, extracted from the AC-3$^{\text{rm}}$ algorithm [9] and extended to handle non-binary constraints (Algorithm 3). *this* denotes the current constraint. τ is a tuple containing a value, denoted $\tau[X]$, for every variable $X \in \text{vars}(C)$. τ is said to be *valid* iff $\forall X \in \text{vars}(C), \tau[X] \in \text{dom}(X)$. The `findSupport` method seeks for an allowed, valid tuple supporting the given value for the current constraint, and returns $\bot$ if no such tuple can be found. If a support is found, it is recorded as a *residue* [11], exploiting the *multidirectionality* of the constraints. A most interesting feature of residues is that they are *stable on backtrack*, that is, when using the MAC procedure, residues that are found at some point of the search tree will also be valid after a backtrack. No update of the data structures is thus necessary upon backtracking.

[1] Many solvers use events to avoid the management of Δ sets.

revise$^{\mathrm{rm}}$ may be considered as the state-of-the-art algorithm to propagate, within MAC and using coarse-grained propagation queues, constraints defined in extension.[2] It can be used as a "fallback" propagator when no better algorithm exists or is implemented yet. Many efficient propagators may also be built on this algorithm, simply by specializing the `findSupport` method: although the standard behavior consists in iterating over all the $O(d^{k-1})$ valid tuples, checking the constraint (in $O(k)$) until an allowed tuple is found, better methods may be devised when working with known constraints. For example, with the $X = Y + Z$ constraint, a support for a value $x \in \mathrm{dom}(X)$ can be found in $O(d)$: the algorithm iterates over the values $y \in \mathrm{dom}(Y)$, and checks whether the value $z = x - y \in \mathrm{dom}(Z)$.

4 Managing the propagation queue

Note: In this paper, all heuristics and sorting algorithms are min-based (minimum value first). Of course, it is perfectly feasible to reverse all comparisons to obtain max-based heuristics and sortings, without any impact on the algorithms and complexities.

4.1 Related work on ordering heuristics

The order in which the constraints are revised has an important impact on the performance of the propagation, and several works have been devoted to devise a good heuristic to know which constraint to propagate first. The original work is by Wallace & Freuder [20]. In their work, they study the impact of various ordering heuristics in an arc-oriented AC-3 propagation algorithm, restricted to binary CSPs. The heuristics devised by Wallace & Freuder follow this principle: for an efficient propagation, values should be filtered as soon as possible, so most constraining constraints should be propagated first.

Of course, it is very difficult to predict how strong a constraint is beforehand. Wallace & Freuder use the tightness (proportion of instantiations forbidden by the constraint) as an heuristic to estimate the strength of the constraint, which is reasonable when working on small binary CSP. However, computing the tightness of a general constraint is $\#P$-hard. Proposed less time-consuming alternatives consider the domain size (we call this the *dom* heuristic) or the degree of the variable in the arc. Note that even when working on tiny binary CSP, Wallace & Freuder's best results were obtained by applying an heuristic before the first propagation (using a pigeonhole sort algorithm), and rely on simple queues or stacks afterwards. An interesting alternative, proposed by Balafoutis & Stergiou in [1], is to exploit the constraint weights obtained from the *dom/wdeg* variable assignment heuristic [5] to devise the most interesting constraints. Moreover, this strategy seems to interact positively with the variable assignment heuristic. Both Wallace & Freuder and Balafoutis & Stergiou works are primarily oriented towards binary CSPs, using plain AC-3 for propagation.

[2] For binary constraints, one can refer to [10] for the revise$^{\mathrm{bit}}$ propagator.

Boussemart *et al.* study and experiment in [6] different revision ordering strategies, using either arc, variable or constraint-oriented propagation queues. As in previously cited works, Boussemart *et al.* perform their experiments on binary CSPs and use an homogeneous propagation algorithm, an improved variant of AC-3. The main result of their work is that the best variant in this context is the variable-oriented propagation scheme with the *dom* variable revision ordering heuristic, a result quite close to Wallace & Freuder's. Indeed, although constraint or arc-oriented revision ordering heuristics (using the product of the size of the domains of the variables in the scope of a constraint, an heuristic we call *Π dom*) successfully reduces the number of constraint checks compared to variable-oriented heuristics, they require a high overhead to compute the heuristics. However, the data structures used by Boussemart *et al.* can be greatly improved.

Another work of interest is [15] by Schulte & Stuckey. The authors explain the propagation scheme implemented at the core of the Gecode solver [14]. The technique is based on another folklore knowledge: since we cannot predict whether a constraint will filter values or not, let us minimize lost time by propagating the fastest constraints first. This technique may only be used with an arc or constraint-based propagation queue. An small integer identifier is associated to each constraint: 0 for very fast constraints (i.e., constant-time or $O(k)$ propagators), 1 for fast constraints (i.e., $O(d)$ propagators), up to 7 for the slowest constraints (i.e., NP-hard propagators). The propagation queue is divided in 8 FIFOs, and the integer identifies the queue in which the constraint is assigned. When picking a constraint for revision, the first FIFO is polled first, then the second if the first is empty, and so on. Moreover, Schulte & Stuckey propose to adapt the identifier dynamically, as even a NP-hard propagator can be applied quite quickly if most variables in the scope of the constraint are assigned.

Interestingly enough, the most successful ordering heuristics devised by Wallace & Freuder or Boussemart *et al.* (*dom* or *Π dom*) also cover the "fastest constraints first" principle: the AC-3-based propagations algorithms used in their experiments use propagators whose time complexities are highly correlated with the size of the domains.

4.2 Fine, constraint-based revision ordering heuristics

Firstly, we give a simple example showing the limits of the variable-based propagation scheme when the CSP include large arity constraints. Let $\mathcal{N}$ be a CSP with n variables X_1 to X_n, $\text{dom}(X_i) = \{1, \dots, n\}$, and the constraints $X_i \leq X_{i+1}$ $\forall i \in \{1, \dots, n-1\}$ and $\text{alldifferent}(X_1, \dots, X_n)$. The *alldifferent* constraint is implemented using an easy algorithm, which filters out all values present in singleton domains, and checks whether $\left|\bigcup_{X \in \text{vars}(C)} \text{dom}(X)\right| \leq |\text{vars}(C)|$. This propagator does not establish (G)AC but is idempotent, detects trivial pigeonhole cases and has a quite low complexity ($O(kd)$).

Let us remove the lowest value from the domain of X_1 and propagate. Using a variable-based propagation scheme with any heuristic, or a constraint-based

Constraint	Evaluator
$X\{<,\leq,>,\geq,\neq\}Y$	2
$\bigvee(\dots)$	$\log_2(\lvert\text{vars}(C)\rvert)$
$\sum_{X\in\text{vars}(C)} X \leq k$	$\lvert\text{vars}(C)\rvert$
alldifferent$(\dots)$	$\lvert\text{vars}(C)\rvert^2$
$a \times X + b = Y$	$\min(\lvert\text{dom}(X)\rvert, \lvert\text{dom}(Y)\rvert)$
$X = Y\{+,\times\}Z$	$\lvert\text{dom}(X)\rvert \lvert\text{dom}(Y)\rvert + \lvert\text{dom}(X)\rvert \lvert\text{dom}(Z)\rvert + \lvert\text{dom}(Y)\rvert \lvert\text{dom}(Z)\rvert$
$X \iff C(\dots)$	$\mathit{evaluator}(C) + \mathit{evaluator}(\neg C)$
positive table	table size $\times \lvert\text{vars}(C)\rvert$
revise$^{\text{rm}}$	$\prod_{X\in\text{vars}(C)} \lvert\text{dom}(X)\rvert$
revise$^{\text{bit}}$ [10]	$\lvert\text{dom}(X)\rvert \lvert\text{dom}(Y)\rvert \div 10$

Table 1. Evaluators for various constraints.

propagation scheme with simple FIFO behavior, the propagator for $X_1 \leq X_2$ is called, removing the lowest value from X_2, then the propagator for *alldifferent*, then the propagator for $X_2 \leq X_3$, then *alldifferent* again, etc. With a constraint-based propagation scheme and a simple heuristic that prioritizes the stronger and faster $\leq$ constraints over *alldifferent*, the propagator for *alldifferent* would be called only once, hence a much faster propagation.

As a reference, our implementation requires 4 s to propagate the above scenario with $n = 1{,}000$ using a variable-based propagation scheme and 50 ms with a prioritized constraint-based propagation scheme.

We define a constraint-based heuristic as follows: each constraint type must implement an *evaluator*, i.e., a method that returns a float number. The number gives an estimation of the time required to propagate the constraint. In our implementation, we use either the average-case complexity if available, or the worst-case complexities of the propagators to compute the estimation. For the general-purpose revise$^{\text{rm}}$ propagator, we fallback to the *Π dom* heuristic. Table 1 summarizes the evaluators we use for the various constraints implemented in our constraint solver. In the remaining of this paper, we will call this constraint revision ordering heuristic *eval*.

We combine our scheme with the ideas from Balafoutis & Stergiou [1], by dividing the value computed by the evaluator by the constraint weight, leading to the so-called *eval/w* constraint revision ordering heuristic.

4.3 Data structures for priority queues: a survey

When using a heuristic for extracting the variable/constraint with the highest/lowest score, the queue is basically a *priority queue*. Various data structures have been proposed in the algorithmic literature for handling these. This section surveys a few of them.

AC-5$^{\text{v}}$ and AC-5$^{\text{c}}$ require two basic operations: inserting an object in the priority queue, and extracting the "best" object, that minimizes the score computed by some heuristic. Upon insertion, if the object is already in the queue, its

Data structure	Insert	Update	Remove min	Heuristics	Plot
m linked lists	$O(1)$	$O(1)$	$O(m)$	FIFO + m levels priority	—+—
Bit vector	$O(1)$	$O(1)$	$\Theta(\lambda)$	Any	—△—
Binary heap	$O(\log \lambda)$	$O(\log \lambda)$	$O(\log \lambda)$	Any	—●—
Binomial heap	$O(1)$*	$O(\log \lambda)$	$O(\log \lambda)$	Any	—○—
Fibonacci heap	$O(1)$	$O(1)$*	$O(\log \lambda)$*	Any	—×—
Soft heap	$O(1)$*	N/A	$O(1)$*	Any (approximated)	—□—

Table 2. Various data structures for implementing priority queues. * denotes an amortized complexity.

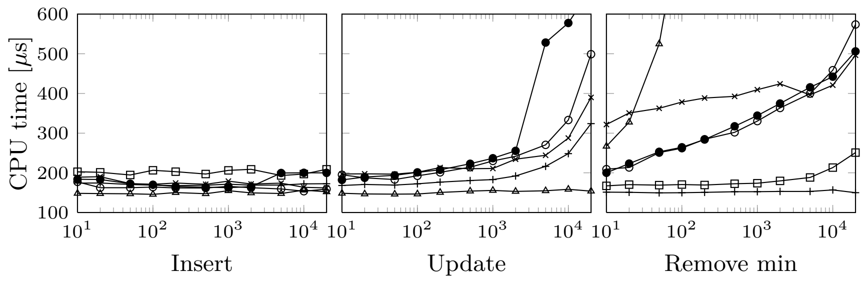

Fig. 1. Actual performance of our implementations: time to insert, update or remove minimum in a set of λ elements. Notice the semilog scale: straight plots are actually log-like.

position is updated. All heuristics devised so far may only evolve during propagation when a variable domain is modified. Thus, it is perfectly sound to update the heuristic score of a constraint only when such an event occur.

In order to experiment the various data structures, all queues implement the generic `Queue` interface as defined in the Java 1.6 API. All queues must be backed by a Set implementation in order to support the Update operation and preventing the same object to be inserted twice.

Here follows the list of data structures we implemented and experimented. Table 2 give the worst-case time complexities for both three basic operations on a structure containing λ elements. Space complexity is $\Theta(\lambda)$ for all structures except multiple linked lists which are in $\Theta(m + \lambda)$. Figure 1 shows the performance of our implementations. We implemented the data structures in Java and benchmarked them using Sun's Java 1.6u21 64-bit HotSpot Virtual Machine for Linux, running on a Intel Core 2 Duo processor @ 2.53 GHz. The JiP 1.2 profiler [21] was used to measure the performance of the various operations, using the following experimental protocol: λ random integer values are inserted in the priority queue. Then the three following operations are performed 1,000,000 times: an additional random integer value is inserted, the minimum integer is extracted from the queue, then an element is randomly updated. The profiler is used to measure the time consumed by each of these three operations. Reported times are for inserting/updating/removing one single element.

Linked lists are the most common way to implement queues in propagation algorithms. Actually, FIFO queues are in the core of most solvers.[3] However, linked lists are not designed to be sorted, and picking the smallest element requires to parse all elements. Using multiple linked lists is an efficient way to implement very coarse heuristics. The heuristic computes an small integer number that identifies a FIFO queue in which the variable/constraint is stored. Our benchmarks used 8 FIFO linked lists. To try to emulate the behavior described in [15], the appropriate FIFO is chosen based on the result of the operation $\lfloor \log_3(h) \rfloor$ (h is the score computed by the heuristic). Indeed, the main factor Schulte & Stuckey use to choose the appropriate FIFO is the arity of the constraint, and the score h computed by the traditional $\Pi\mathit{dom}$ heuristic is in $O(d^k)$. The base 3 was chosen in order to normalize the use of all 8 FIFOs in the average case. When an update is requested, the variable/constraint is moved to the tail of the appropriate FIFO if needed.

Bit vectors can replace linked lists when static or heuristic ordering is used. One bit is associated to each variable or constraint. The bit is set upon insertion, and cleared upon removal. This ensures very fast insertion and update operations, but finding the minimal element still requires to parse all elements. Boussemart *et al.* used this scheme in their paper [6].

Binary heap is a well known data structure, and can be used for implementing priority queues. A binary heap is a naturally balanced binary tree in which each parent node is smaller than its children. The smallest element is thus always at the root of the tree. Inserting, updating and removing an element requires $O(\log \lambda)$ *sift* operations to maintain the heap property.

Binomial heap [19] use a special tree structure (a "forest" of heap trees) to achieve fast insertions, although removing the smallest element has the same performance as insertion. Updates are basically performed by removing and reinserting the element.

Fibonacci heap [7] is a "lazy" variant of binomial heaps, in which most computations are delayed until the remove min operation is called. Insertions and updates are thus very fast, although the remove min operation is not much slower than with binomial heaps, and even more robust for large amounts of data.

Soft heap [8] is a variant of binary heap in which the heap property is only maintained on the top of the tree. The root of the tree is thus no longer guaranteed to be the smallest element. However, the "corruption" is minimal and can be parameterized (we used $\epsilon = 10\,\%$). This permits constant $O(\log \frac{1}{\epsilon})$ complexities for both insert and remove min operations. However, insertion is noticeably slower than with other data structures, and update is not supported by our implementation (the element is simply left in place upon updating).

[3] Simple experiments show that LIFO strategies are almost always worse than FIFO.

			AC-5^v			AC-5^c		
	n	e	Inserts	Updates	Remvs	Inserts	Updates	Remvs
bqwh-18-141-0-ext	141	879	4.0 M	647 k	2.2 M	21 M	4.3 M	14 M
bqwh-18-141-47-glb	141	36	29 M	6.2 M	15 M	28 M	7.7 M	20 M
frb40-19-1	40	410	2.3 M	1.4 M	1.3 M	19 M	16 M	14 M
series-18	69	36	3.1 M	963 k	2.5 M	3.5 M	2.1 M	3.3 M
ruler-44-9-a3	45	74	2.9 M	1.5 M	2.0 M	4.5 M	6.0 M	3.3 M
langford-3-13	65	27	11 M	6.0 M	8.3 M	9.0 M	15 M	7.4 M
bmc-ibm-02-02	50 k	48 k	91 k	23 k	91 k	94 k	23 k	94 k
crossword-m1-lex-15-04	4.4 k	7.9 k	12 M	569 k	11 M	167 M	8.0 M	167 M
lemma-24-3	552	924	20 M	0	14 M	57 M	2.8 M	47 M
os-taillard-5-100-3	625	500	66 M	6.1 M	59 M	125 M	15 M	103 M
scen4	8.3 k	7.6 k	55 k	24 k	53 k	95 k	47 k	92 k
bigleq-70	70	70	18 M	16 M	10 M	30 M	36 M	19 M

Table 3. Number of operations required to solve various problems with the *eval* heuristic.

Choosing the best data structure may depend on the number of elements it will contain, as well on the relative importance of the insert, update and remove operations. As a reference, the problems used as benchmarks during the CPAI'08 Itl Solver Competition [16] had on average 863 explicit variables (from 2 to 62,704, std dev is 3,100, median 120) and 5,129 explicit constraints (from 1 to 546,105, std dev is 20,065, median 458). Moreover, using techniques such as constraint decomposition, symmetry breaking, implicit constraint detection, second-order consistencies or nogood learning can increase the number of variables and/or constraints significatively.

Table 3 gives an idea of the relative number of operations required to solve some well-known benchmark problems using the *eval* revision heuristic and the *dom*/*ddeg* decision heuristic (the more efficient *dom*/*wdeg* heuristic was not used to avoid any interference with the revision heuristic). The second and third columns, n and e, respectively show the number of variables and constraints actually present in the problem once the solver has performed appropriate decompositions. The number of removals is usually less than the number of inserts because the propagation is interrupted (and the priority queues cleared) when an inconsistency is encountered.

4.4 Note on Set implementation

Several data structures can be used to implement sets: hashtables, ordered trees, etc. For best, constant-time performance, we rely on simple arrays. An contiguous integer identifier is associated to each object upon creation, which identifies the index of the array where the structures will be stored. A basic set implementation can thus use an array of booleans (or a bit vector).

Clearing the sets is also an operation that can have a non negligible impact on the performances of the resolution. In some of our experiments, an $O(\lambda)$

	Bit vector	8 FIFOs	Bin Heap	Binom H	Fib Heap	Soft Heap
bqwh-18-141-0-ext	20.0 s	11.6 s	26.3 s	11.4 s	12.0 s	14.8 s
bqwh-18-141-47-glb	35.4 s	32.6 s	35.6 s	34.3 s	34.8 s	35.4 s
frb40-19-1	27.8 s	11.8 s	23.0 s	11.0 s	12.4 s	13.7 s
series-18	5.7 s	5.0 s	0.4 s	4.8 s	4.9 s	4.9 s
ruler-44-9-a3	8.3 s	7.2 s	27.1 s	6.8 s	7.0 s	6.8 s
langford-3-13	15.1 s	13.7 s	1.9 s	14.2 s	14.3 s	13.7 s
bmc-ibm-02-02	339.0 s	20.2 s	20.3 s	20.5 s	20.8 s	20.0 s
crossword-m1-lex-15-04	2,204.3 s	210.1 s	433.0 s	265.0 s	273.8 s	283.0 s
lemma-24-3	85.0 s	74.4 s	113.0 s	82.9 s	87.8 s	96.2 s
os-taillard-5-100-3	1,579.6 s	205.7 s	113.3 s	188.9 s	199.0 s	291.7 s
scen4	11.3 s	6.2 s	7.1 s	6.4 s	6.7 s	6.9 s
bigleq-70	318.9 s	45.8 s	50.3 s	42.9 s	43.8 s	42.0 s

Table 4. Time to solve the problems using AC-5^c and *eval* heuristic with various priority queues.

clear operation could take more than 90 % of the CPU time required to solve the problem! Set clearing can be performed in $O(1)$ by using integer counter, as proposed in [6] for the *modified* data structure. An integer number i is associated to the set, and is incremented when clearing is requested. When an object O is put in the set, the number $i_O = i$ is stored in the structure representing the object. The object is considered to be present in the set iff $i_O = i$.

5 Experiments

These experiments are performed in the same conditions as before (Sun's Java 1.6u21 64-bit HotSpot Virtual Machine for Linux, running on a Intel Core 2 Duo processor @ 2.53 GHz), but without the use of a profiler. The constraint solver used is CSP4J [18]. We selected representative problem instances from the CPAI'08 competition, that could be solved between 2 and 300 s using the *dom/ddeg* decision heuristic. Our experiments are still preliminar: our solver only implements a few constraint types, and the problem base of CPAI'08 lacks challenging problems with global constraints, which reduces the "heterogeneity" of the selected problems. In particular, few of them use global constraints at all.

A first set of experiments, summarized on Table 4, compares the different data structures using the plain *eval* constraint revision ordering heuristic. These results tend to show that either multiple FIFOs or Binomial heaps are the most efficient data structures for handling constraint revision ordering heuristics (and that linear-time data structures such as bit vectors definitively are not, despite their very fast insert and update operations).

Finally, Table 5 compares the different variable- and constraint-based heuristics. AC-5^v uses a Binomial heap in these experiments. The *dom/ddeg* decision heuristic was used, but constraint weights are still computed as for the *dom/wdeg* decision heuristic. These weights can thus be used for the *dom/wdeg* variable

	AC-5^v		AC-5^c/8 FIFOs				AC-5^c/Binomial heap			
	dom	$\frac{dom}{wdeg}$	*Π dom*	$\frac{\Pi dom}{w}$	*eval*	$\frac{eval}{w}$	*Π dom*	$\frac{\Pi dom}{w}$	*eval*	$\frac{eval}{w}$
bqwh-18-141-0-ext	9.0	9.2	8.9	11.7	10.8	10.9	10.4	11.4	10.5	11.5
bqwh-18-141-47-glb	32.5	32.3	33.8	33.8	33.4	33.7	35.1	33.8	36.2	33.0
frb40-19-1	7.8	8.7	9.7	12.7	12.0	13.2	12.1	14.2	12.2	14.2
series-18	4.9	5.3	5.2	5.1	5.0	5.2	5.1	5.0	5.5	5.5
ruler-44-9-a3	8.9	11.2	7.6	9.1	8.4	10.2	7.2	7.3	7.3	7.8
langford-3-13	17.1	18.1	15.3	15.3	14.4	16.2	15.4	15.9	14.6	16.7
bmc-ibm-02-02	19.6	20.5	19.8	20.0	19.7	19.1	19.6	19.5	19.3	19.2
crosswd-m1-lex-15-04	152.0	139.3	190.6	175.2	220.8	190.9	219.5	187.0	262.6	204.4
lemma-24-3	80.2	85.6	97.7	96.0	91.8	88.6	86.8	89.1	87.0	85.9
os-taillard-5-100-3	206.4	224.1	249.5	190.4	212.6	232.0	257.1	197.9	211.4	205.1
scen4	7.0	6.7	7.4	7.7	7.8	6.4	6.2	6.7	7.0	6.6
bigleq-70	92.4	100.9	34.5	34.5	28.2	97.8	30.8	31.4	27.6	91.2

Table 5. Time (in seconds) to solve the problems with the *dom/ddeg* decision heuristic and different revision ordering heuristics.

and for *Π dom/w* or *eval/w* constraint revision ordering heuristics. Following Balafoutis & Stergiou results described in [1], these ordering heuristics are more senseful when combined with the *dom/wdeg* decision heuristic.

Although we are aware that these experiments still fail to demonstrate a clear superiority of constraint-based heuristics, we are convinced that (1) constraint-based propagation is actually competitive w.r.t. variable-based propagation, (2) it successfully avoids pathological cases (our *bigleq* problem), and (3) opens a new field of research to devise better heuristics.

6 Conclusion & Perspectives

In this paper, we devised AC-5^v and AC-5^c, generic coarse-grained propagation algorithms using respectively variable- and constraint-based propagation queues. After recalling why the management of the propagation queue is important, we surveyed a few data structures that can be used to control the order in which variables or constraints will be revised.

We proposed a new, generic way to control the order in which the constraints are revised using the constraint-based propagation scheme, and showed experimentally that using clever data structures, this way of controlling the propagation can be competitive w.r.t. variable-based propagation, and can avoid pathological cases. These cases will occur frequently when using heavy global constraints, such as NP-hard constraints introduced by Lazy Clause Generation or algorithm hybridization. Variable-based propagation will then no longer be a viable alternative.

Although our heuristics are still not clearly better than standard general heuristics, we hope to open the perspectives to devise new techniques, either adaptative or by taking into account the strength of the constraints.

References

1. T. Balafoutis and K. Stergiou. Exploiting constraint weights for revision ordering in arc consistency algorithms. In *Proceedings of the ECAI-2008 workshop on Modeling and Solving Problems with Constraints*, 2008.
2. N. Beldiceanu and E. Contejean. Introducing global constraints in CHIP. *Mathl. Comput. Modelling*, 20(12):97–123, 1994.
3. C. Bessière and J.-C. Régin. Arc consistency for general constraint networks: preliminary results. In *Proceedings of IJCAI'97*, pages 398–404, 1997.
4. C. Bessière, J.-C. Régin, R.H.C. Yap, and Y. Zhang. An optimal coarse-grained arc consistency algorithm. *Artificial Intelligence*, 165(2):165–185, 2005.
5. F. Boussemart, F. Hemery, C. Lecoutre, and L. Saïs. Boosting systematic search by weighting constraints. In *Proceedings of ECAI'04*, pages 146–150, 2004.
6. F. Boussemart, F. Hemery, and C. Lecoutre. Revision ordering heuristics for the Constraint Satisfaction Problem. In *Proceedings of CPAI'04 workshop held with CP'04*, pages 29–43, 2004.
7. M.L. Fredman and R.E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615, 1987.
8. H. Kaplan and U. Zwick. A simpler implementation and analysis of chazelle's soft heaps. In *Proc. of the 19th ACM-SIAM Symposium on Discrete Algorithms*, pages 477–485, 2009.
9. C. Lecoutre and F. Hemery. A study of residual supports in arc consistency. In *Proceedings of IJCAI'2007*, pages 125–130, 2007.
10. C. Lecoutre and J. Vion. Enforcing Arc Consistency using Bitwise Operations. *Constraint Programming Letters*, 2:21–35, 2008.
11. C. Likitvivatanavong, Y. Zhang, J. Bowen, and E.C. Freuder. Arc consistency in MAC: a new perspective. In *Proceedings of CPAI'04 workshop held with CP'04*, pages 93–107, 2004.
12. A.K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977.
13. J.J. McGregor. Relational consistency algorithms and their application in finding subgraph and graph isomorphisms. *Information Sciences*, 19:229–250, 1979.
14. C. Schulte, M. Lagerkvist, G. Tack, et al. Generic Constraint Development Environment (Gecode). `http://www.gecode.org/`, 2005-2010.
15. C. Schulte and P.J. Stuckey. Efficient Constraint Propagation Engines. *ACM Transactions on Programming Languages and Systems*, 31(1):1–43, 2008.
16. M. van Dongen, C. Lecoutre, and O. Roussel. Third International CSP Solvers Competition. `http://www.cril.univ-artois.fr/CPAI08`, 2008.
17. P. van Hentenryck, Y. Deville, and CM. Teng. A generic arc-consistency algorithm and its specializations. *Artificial Intelligence*, 57:291–321, 1992.
18. J. Vion. Constraint Satisfaction Problem for Java. http://cspfj.sourceforge.net/, 2006.
19. J. Vuillemin. A Data Structure for Manipulating Priority Queues. *Communications of the ACM*, 21:309–314, 1978.
20. R.J. Wallace and E.C. Freuder. Ordering heuristics for arc consistency algorithms. In *Proceedings of NCCAI'92*, pages 163–169, 1992.
21. A. Wilcox and P. Hudson. Java Interactive Profiler. `http://jiprof.sourceforge.net/`, 2005–2010.

Author Index