Robust constraint solving using multiple heuristics*

Student: Alfio Vidotto[1]

Supervisors: Kenneth N. Brown[1], J. Christopher Beck[2]

[1] Cork Constraint Computation Centre,
Dept of Computer Science, UCC, Cork, Ireland
av1@student.cs.ucc.ie, k.brown@cs.ucc.ie

[2] Toronto Intelligent Decision Engineering Laboratory,
Dept of Mechanical and Industrial Engineering, University of Toronto, Canada
jcb@mie.utoronto.ca

1 Introduction

Representing and solving problems in terms of constraints can be difficult to do effectively. A single problem can be modeled in many different ways, either in terms of representation or in terms of the solving process. Different approaches can outperform each other over different problem classes or even for different instances within the same class. It is possible that even the best combination of model and search on average is still too slow across a range of problems, taking orders of magnitude more time on some problems than combinations that are usually poorer. This fact complicates the use of constraints, and makes it very difficult for novice users to produce effective solutions. The modeling and solving process would be easier if we could develop robust algorithms, which perform acceptably across a range of problems.

In this paper we present one method of developing a robust algorithm. We combine a single model and a single basic algorithm with a set of variable and value ordering heuristics, in a style similar to iterative deepening from standard AI search. The aim is to exploit the variance among the orderings to get a more robust procedure, which may be slower on some problems, but avoids the significant deterioration on others. During the search, we allocate steadily increasing time slices to each ordering, restarting the search at each point. We demonstrate its performance on two problem classes, showing that it is robust across problem instances and competitive with standard orderings used for those problems.

2 Background

The standard process for generating solutions to a CSP is based on backtracking search. The order in which variables and values are tried has to be specified as part of the search algorithm, and has a significant effect on the size of the search tree. The standard ordering heuristic is based on the “fail-first” principle, stating that we should

* funded by Enterprise Ireland (SC/2003/81), with assistance from Science Foundation Ireland (00/PI.1/C075) and ILOG, SA.

choose the variable with the tightest constraints. This is normally implemented by choosing the variable with the minimum domain, or the smallest ratio of domain size to degree. Strategies aiming to "succeed first" have also been investigated, e.g. in [1] where different variable heuristics showed different search efforts, depending on their level of "promise". For an instance of a CSP, a single run with a single ordering heuristic can get trapped in the wrong area of the tree, even if the heuristic is the best on average. For this reason, the randomized restart strategy has been proposed - for a single heuristic, if no result has been found up to a given time limit, the search is started again. Tie breaking and, typically, value ordering are done randomly, and so each restart explores a different path. This approach is known to work well on certain problems, including quasi-group with holes [2]. Algorithm portfolios [3] is another randomized restart search method, which interleaves a set of randomized algorithms.

3 Multi-heuristic and time-slicing

As discussed above, for many problem classes no single ordering heuristic performs well across all problem instances. In some initial experiments on a scheduling problem, we had noticed that some instances caused a 1000-fold increase in running time. Further, the hard instances appeared to be different for each ordering. Therefore, we have developed an approach which tries each ordering in turn for a limited time, restarting the search after each one, and gradually increasing the time limit if no result was found. This is similar to the way iterative deepening explores each branch to a certain depth, and then increases the depth limit, and is similar to randomized restarts, except we use different ordering heuristics. The pseudo code for the multi-heuristic (MH) algorithm is:

```
while Solve(heuristic(i),limit) == false
   limit = Increase(i,limit)
   if  i == n then i = 1
  else  i = i + 1
```

Solve(.,.) takes heuristic *i* (composed of a variable and a value ordering), and applies standard search up to a time *limit*. If it finds a solution, or proves there is no solution, it returns *true*; otherwise it hits the time limit and returns *false*.

Increase(.,.) is the time limit function. We have considered two versions: (*linear*) *Increase(i,limit)=limit+δ* and (*magnitude*) *Increase(i,limit)=limit*10 if i=n; limit otherwise*.

Note that MH is complete: the CSP backtracking search space is finite, each ordering heuristic is systematic, and *limit* increases indefinitely, so eventually one of the heuristics will be given enough time to complete the search. Further, if any one of the heuristics is deterministic, then MH has a guaranteed upper bound on the ratio of the time it takes compared to that heuristic.

4 Experiments and results

We want to test the performance of the time-sliced multi-heuristic approach. Specifically, (i) is it more robust than the standard default ordering heuristic, i.e. does it

report a result within acceptable time limits in more cases across a range of problems? (ii) does it avoid a significant increase in run time, i.e. is the overhead of restarting the search, and repeating some search paths, significant? (iii) how does it compare to the randomized restart method, i.e. is its performance due to the restart mechanism, or to the multiple heuristics? To answer these questions, we have tested the approach on two problem classes: scheduling tasks with fixed start and end points, and quasi-groups with holes. All implementations are coded in C++ using Ilog Solver 6.0, and run on a Pentium 2.6 GHz processor under Linux. For (i) and (ii), we compare MH against the min domain (*msd*) variable ordering heuristic (lexicographic tie breaking). For (iii) we use the same *msd* but with random tie breaks, and random value ordering.

Scheduling - We considered one class of scheduling problems, where tasks have fixed start and end times, but can be allocated to a number of different resources. We assume that resources come in categories, and that categories are ranked. Each task has a rank, and must be allocated to a resource of that rank or higher. Each resource can process one task at a time, and each task must be processed without interruption on a single resource. Given a set of resources and tasks, the problem is to determine whether or not the tasks can be scheduled. This problem is known to be NP-complete [4]. In our model, we represent the tasks as variables, and the resources as the values to be assigned, and the constraints ensure tasks do not overlap. We consider one set of test problems, <100, 10, N>, with 100 resources in 10 classes. We varied the number of tasks, N, from 130 to 200 (in single steps), and for each one we generated 500 random problems, choosing start times in [0..40], durations in [17..25] and ranks in [1..10], all uniformly at random. For each instance, we impose a maximum time of 41 seconds, which allows time slices of 0.01, 0.1, 1 sec for 33 possible heuristics, including the overhead on initializing the problem.

Task	Rank	Start	End
T1	3	0	2
T2	2	0	2
T3	3	1	3
T4	1	2	4

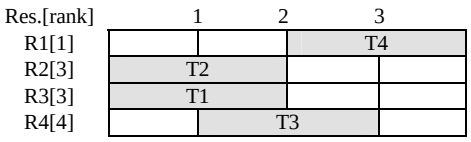

Fig. 1. Scheduling: tasks with ranks, fixed start and end times *(left)*; a possible solution *(right)*.

We combined a list of variable orderings (H1...H11), and a list of value orderings (W1…W3), getting an algorithm we call MH(11x3). H1 and H2 are versions of *msd*, breaking ties randomly (H1), and lexicographically (H2). H3 to H10 are created from sorting the tasks by start time *s* and min resource class *m*, in all combinations (i.e. H3 increasing *s* breaking ties by increasing *m*, .., H10 decreasing *m* breaking ties by decreasing *s*). H11 involves a measure of contention among tasks. It sorts by counting, for each task, the number of other tasks which it overlaps in time (e.g. in Fig. 1, T3 is the most overlapping task and would be the first choice). The value heuristic W1(W2) orders the resources by increasing (decreasing) class, while W3 is a random order.

Quasi-group with holes (QWH) - A quasi-group of order N is a Latin Square (LS) of N by N cells. The solution of a LS requires an allocation to each cell of a number from 1 to N, so that all the numbers appearing on each row are different and all the numbers appearing on each column are also different. A QWH is a solved LS from

which some allocations are deleted. The problem is to find an allocation which completes the LS. We represent the cells as variables, and the numbers as the values to be assigned. We use the Ilog global constraint IloAllDiff on each row and column.

1		2	
	2		

1	3,4	2	3,4
3,4	2	1,3,4	1,3,4
2,3,4	1,3,4	1,3,4	1,2,3,4
2,3,4	1,3,4	1,3,4	1,2,3,4

1	3	2	4
3	2	4	1
2	4	1	3
4	1	3	2

Fig. 2. Quasi group with holes: an instance, remaining domains, and a solution.

We utilized a list of variable orderings (H1...H10), and a list of value orderings (W1…W3), (MH(10x3)). H1 and H2 are versions of *msd*, breaking ties randomly (H1), and lexicographically (H2). H3 to H10 are created by sorting the cells by column *c* and row *r*, in all combinations (i.e. H3 is increasing *c* breaking ties by increasing *r*, .., H10 is decreasing *r* breaking by decreasing *c*). The value heuristic W1(W2) simply chooses smallest (biggest) number first, while W3 uses a measure of conflict among numbers. If variable X is chosen, W3 looks the number frequency in the domains of the unassigned variables in the same row and column as X. Knowing that all numbers must appear once in the column and once in the row, W3 chooses the one that appears least in domains of the unassigned variables in the row and column (e.g. assuming X = bottom right cell in Fig.2, then W3 would select number 2).

Comparing to min domain on scheduling – In Fig.3 we show the number of times *msd* and MH(11x3) hit the time limit, and the mean run time. MH consistently outperforms and improves *msd*. It is more robust – it hits the time limit on fewer occasions. It also has a lower mean run time across the range. Note that the line on the graph from top left to bottom right shows solubility, and relates to the right hand axis – e.g. almost 50% of size 150 problems have a solution. The hardness peak is where most problems have no solution.

failure frequency [%]		
Size [N]	*msd*	MH magnitude
130	10	4
140	22	12
150	62	16
160	58	32
170	82	40
180	28	22
190	2	0
200	0	0

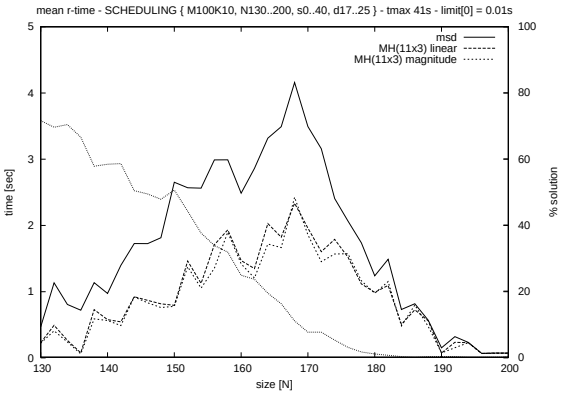

Fig. 3. MH vs. SH: left, frequency [%] of failure to solve within *t-max*; right, mean r-time;

Comparing to min domain on balanced QWH – In Fig. 4, we again show robustness and run time, this time for balanced QWH(20). MH(10x3) again consistently outperforms *msd* both in terms of robustness and time. All problems have solutions.

failure frequency [%]		
Size [H]	*msd*	MH magnitude
150	0	0
170	70	20
190	100	50
210	60	20
230	70	0
250	60	0
270	30	0
290	20	0

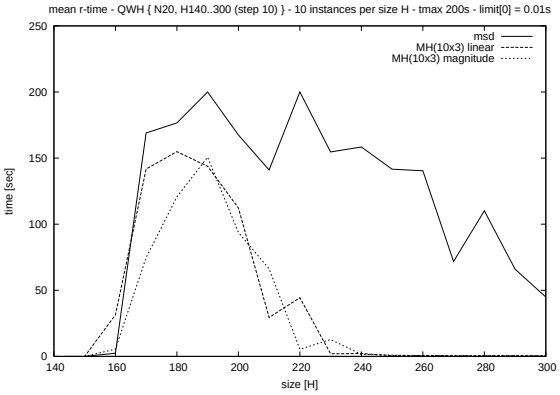

Fig. 4. MH vs. SH: left, frequency [%] of failure to solve within *t-max*; right, mean r-time;

Comparing to randomized-restarts (RR) - RR is regarded to be the best method for QWH. We have compared MH with RR on both QWH and scheduling. RR is generally used with time limits that increase each restart, so we have implemented MH with the same time policy, and RR with an order of magnitude time increased every N restarts, for comparison.

QWH *(Fig. 5)* – RR is better than MH almost everywhere, regardless of which time slicing mechanism we use. Both MH and RR performed slightly better with time slices increased by a magnitude every loop of restarts, for which we report the statistic on the frequency of failure.

failure frequency [%]		
Size [H]	RR magnitude	MH magnitude
150	0	0
160	0	0
170	30	20
180	40	50
190	20	50
200	10	30
210	0	20
220	10	0

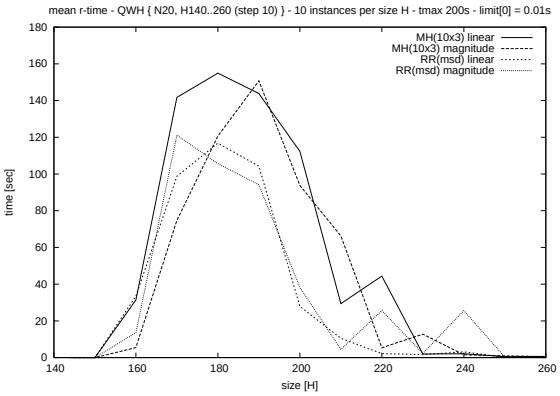

Fig. 5. MH vs. SH: left, frequency [%] of failure to solve within *t-max*; right, mean r-time;

Scheduling *(Fig. 6)* **-** MH clearly reduces the peak of difficulty, which is located in the region where approximately 90% of instances have no solution. The gap is present for both slicing versions, with the magnitude mechanism better on average.

failure frequency [%]		
Size [N]	RR magnitude	MH magnitude
130	2	4
140	14	12
150	18	16
160	42	32
170	62	40
180	32	22
190	0	0
200	0	0

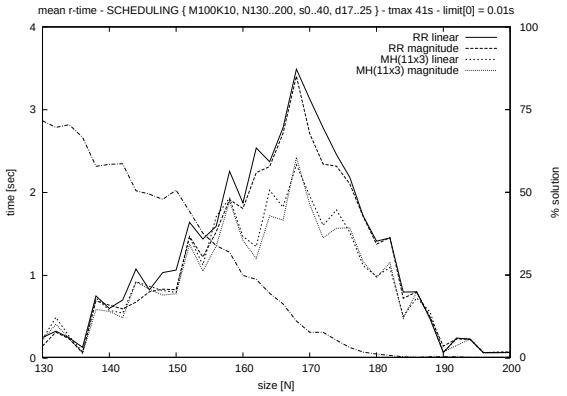

Fig. 6. MH vs. SH: left, frequency [%] of failure to solve within *t-max*; right, mean r-time;

5 Conclusions and future work

We have developed a multi-heuristic approach for constraint solving, designed to improve search robustness. We have tested it on two problem classes, and shown that it is more robust than the standard recommended heuristic, without decreasing run time – in fact, on average it improves the run time. We have also compared to randomized restarts, the leading method for one of our problem classes (QWH) and which uses a similar restart policy. We have shown that the multi heuristic approach is poorer in run time and robustness on QWH, but better on our scheduling problem class. Note that the different heuristics we use and the different time limits have not been tuned – they were generated by inspection of the problem characteristics, and better performance should be achievable. For the immediate future, we intend to investigate whether MH does perform better on insoluble problems (as indicated by the scheduling results).

We can conclude that the multi heuristic method offers a robust and competitive approach to constraint solving, and merits further investigation, since it offers one possible solution to the goal of making CP easier to use.

References

1 Beck, J. C; Prosser, P; and Wallace, R. J. Variable Ordering Heuristics Show Promise, *Proceedings of the Tenth International Conference on Principles and Practice of Constraint Programming* (CP'04), 2004.

2 Gomes, C. P.; and Shmoys, D. B. 2004. Approximations and Randomization to Boost CSP Techniques. In *Annals of Operation Research*, 130:117-141.

3 Gomes, C. P.; and Selman, B. 2001. Algorithm portfolios. In *Artificial Intelligence* 126(1-2):43-62.

4 Arkin, E. M.; and Silverberg, E. B. 1987. Scheduling jobs with fixed start and end times. In *Discrete Applied Mathematics*, 18:1-8.