Robust constraint solving using multiple heuristics

Alfio Vidotto[1], 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

Abstract. Constraint Programming is a proven successful technique, but it requires skill in modeling problems, and knowledge on how algorithms interact with models. What can be a good algorithm for one problem class can be very poor for another; even within the same class performance can vary wildly from one instance to another. CP could be easier to use if we could design robust algorithms that perform well across a range of problems, models and instances. In this paper we look specifically at variable and value ordering heuristics for backtracking search and propose a multi-heuristic algorithm based on time-slicing, and we demonstrate its performance on two different problem classes, showing it is more robust than the standard heuristics.

1 Introduction

Constraint Satisfaction is a proven AI technique, with many successful and profitable applications. However, 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 search 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 different problem classes, showing that it is robust across problem instances, and is competitive with standard orderings used for those problems.

2 Background

A Constraint Satisfaction Problem (CSP) is defined by a set of decision variables, $\{X_1, X_2, ..., X_n\}$, with corresponding domains of values $\{D_1, D_2, ..., D_n\}$, and a set of constraints, $\{C_1, C_2, ..., C_m\}$. Each constraint is defined by a scope, i.e. a subset of variables, and a relation which defines the allowed tuples of values for the scope. A state is an assignment of values to some or all of the variables, $\{X_i = v_i, X_j = v_j, ...\}$. A solution to a CSP is a complete and consistent assignment, i.e. an assignment of values to all of the variables, $\{X_1 = v_1, X_2 = v_2, ..., X_n = v_n\}$, that satisfies all the constraints.

The standard process for generating solutions to a CSP is based on backtracking search. This proceeds by selecting a variable and then choosing a value to assign to it. After each assignment, it propagates the constraints by removing inconsistent values from the domains of future variables. If none of the future domains are empty then search continues by selecting another variable; otherwise it backtracks, selects another value from the domain of the current variable and continues; if no other values are possible, it backtracks to the previous variable. 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 so called "fail-first" principle, stating that we should choose the variable with the tightest constraints. This is normally implemented by choosing the variable with the smallest remaining domain, or the smallest ratio of domain size to degree (representing the CSP as a graph, with variables as nodes and constraints as edges). Strategies aiming to "succeed first" have also been investigated, e.g. in [4] where different variable heuristics showed different search efforts, depending on their level of "promise". Even the choice of a value ordering heuristic represents an important aspect in setting up a good search algorithm. Among the most effective for many CSPs is the min-conflicts value heuristic [5], which chooses the value that rules out the fewest choices for the neighboring variables in the constraint graph. The reason why ordering heuristics matter is because if the search makes a bad choice at the top of the search tree, it can waste a lot of effort exploring sub-trees that have no solution. In [8], the behavior of standard variable ordering heuristics over insoluble sub-trees is compared to optimal refutations, with the advice that some knowledge on how refutations distribute may be relevant to improve the search.

For a single 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 has been shown to work well on certain problems, including quasi-group with holes [7]. Algorithm portfolios [6] is another randomized restart search method, which interleaves a number 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 in comparison to others. 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 pseudocode 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(.,.) simply takes heuristic *i* (composed of a variable ordering 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. Secondly, 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

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 (QWH).

All implementations are coded in C++ using Ilog Solver 6.0, and run on a Pentium 2.6 GHz processor under Linux. In each case we compare our multi-heuristic approach against the recommended heuristics. For (i) and (ii), we compare MH against the smallest remaining domain (*msd*) variable ordering heuristic (with lexicographic tie breaking). For (iii) we compare against the same variable ordering heuristic but with random tie breaks, and random value ordering.

4.1 Scheduling

The problem - 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 categorized resources and ranked tasks, with fixed start and end times, the problem is to determine whether or not the tasks can be scheduled. This problem is known to be NP-complete [2]. 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.

Example - In Fig. 1 we represent: four tasks with rank, and fixed start and end times (*left*); and a possible solution (*right*).

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

Res.[rank]	1	2	3	
R1[1]			T4	
R2[3]	T2			
R3[3]	T1			
R4[4]		T3		

Fig. 1. Scheduling tasks with fixed start and end times over ranked resources

Variable orderings - We utilized the list of variable (task) orderings represented in Table 1. H1 and H2 are two standard versions of min-size domain. H3 to H10 are static orderings created from sorting the set of tasks by start time and minimum resource class. H11 involves a measure of time contention [3] among tasks, i.e. it sorts by counting, for each task, the number of other tasks which overlaps in time. Thus, in the example of Fig. 1, task T3 would count 3 (overlapping with T1, T2, T4), T1 and T2 would count 2, and T4 would count 1, so T3 would be tried first.

Value orderings - We utilized the list of value (resource) orderings represented in Table 2, including three static orders: two choose among resources with the smallest or highest (suitable) class first; one consisting of a random resource order.

Table 1. List of variable ordering heuristics

Heuristic id	Ordering	Tie breaking
H1	min size domain	random
H2	min size domain	lexicographic
H3	increasing start time	increasing min-resource class
H4	increasing start time	decreasing min-resource class
H5	decreasing start time	increasing min-resource class
H6	decreasing start time	decreasing min-resource class
H7	increasing min-resource class	increasing start time
H8	increasing min-resource class	decreasing start time
H9	decreasing min-resource class	increasing start time
H10	decreasing min-resource class	decreasing start time
H11	most overlapping in time	lexicographical

Table 2. List of value ordering heuristics

Heuristic id	Ordering	Tie breaking
W1	min class resource first	lexicographic
W2	max class resource first	lexicographic
W3	arbitrary fixed order	

Multi heuristic approach - We combined both lists of variable and value heuristics together, implementing four different MH versions: MH(11x3), MH(11x1), MH(1x3), and MH(1x1), all with H1 and W1 as first variable and value heuristics.

Test setting - We consider one set of test problems, $\langle$ 100, 10, N $\rangle$, 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, and 1 second for 33 possible heuristics, including the overhead on initializing the problem.

4.2 Quasi-group with holes

The problem - A quasi-group of order N is a Latin Square of N by N cells. The solution of a Latin Square requires an allocation to each cell of a number from 1 to N, so that all the elements appearing on each row are different and all the elements appearing on each column are also different. A quasi-group with holes (QWH) is a solved Latin Square from which some allocations are deleted. The problem is to find an allocation which completes the Latin Square. In our model, the variables are the empty square cells and the values are the elements to be assigned. In our model, we represent the empty cells as variables, and the numbers as the values to be assigned. We use the Ilog global constraint IloAllDiff to ensure each row and column has allocations that are all different.

Example - In Fig. 2 we represent: a problem instance of QWH(N=4) with H=13 holes (*left*); the remaining domains (*centre*); and a possible solution (*right*).

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

Variable orderings - We utilized the list of variable (cell) orderings represented in Table 3. H1 and H2 are two standard versions of min-size domain. H3 to H10 are static orderings created from sorting the square cells by column and row.

Table 3. List of variable ordering heuristics

Heuristic id	**Ordering**	**Tie breaking**
H1	min size domain	random
H2	min size domain	lexicographic
H3	increasing column	increasing row
H4	increasing column	decreasing row
H5	decreasing column	increasing row
H6	decreasing column	decreasing row
H7	increasing row	increasing column
H8	increasing row	decreasing column
H9	decreasing row	increasing column
H10	decreasing row	decreasing column

Value orderings - We utilized the list of value (number) orderings represented in Table 4, which includes three orders, two static and one dynamic. W1 (W2) simply chooses smallest (biggest) numbers first. W3 involves a measure of conflict among numbers: if variable X is chosen, W3 looks the number frequency in the domains of all 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 choose the number that appears least in domains of the other unassigned variables in the row and column. Thus, in the example of Fig. 2 above, assuming the bottom-right cell (variable) is chosen first, number 1 would count 4, number 2 would count 2, and numbers 3 and 4 would count 6, so W3 would choose number 2 first.

Table 4. List of value ordering heuristics

Heuristic id	Ordering	Tie breaking
W1	min number first	lexicographic
W2	max number first	lexicographic
W3	least (x, y)-conflicted number	lexicographic

Multi heuristic approach - Similarly to the way we proceeded for the scheduling problem, we combined both lists of variable and value heuristics together, implementing MH(10x3), with H1 and W1 as first variable and value heuristics.

Test setting - Experiments regarded balanced QWH problems of order N=20. We used the Gomes generator [1] and generated 10 balanced instances for problems with H holes, and did it for a series of different H around the difficulty peak. On each instance, each algorithm had a limited time length t-max of 200 seconds to solve, after that we considered the run as failed.

5 Results

5.1 Comparing to the standard recommended heuristic (*msd*)

Scheduling - In Fig. 3 we show the number of times msd and MH(11x3) hit the time limit, and the mean run time of all 4 versions. MH in its full 11x3 version consistently outperforms and improves msd (i.e. 1x1). It is more robust - it hits the time limit on fewer occasions. It also has a lower mean run time across the range. We can also observe that, as we start introducing more than one value heuristic, i.e. 1x3, we obtain a first clear improvement. There is benefit also by including more than one variable heuristic, i.e. 11x1. Note that passing from 1x1 to 11x1 the majority of the variable heuristics we add to the dynamic msd are all static. Things get better again when combining all the variable heuristics with all the value heuristics, i.e. 11x3. On other tests we also saw that excluding msd from the full version 11x3 has only a small effect on performance. 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

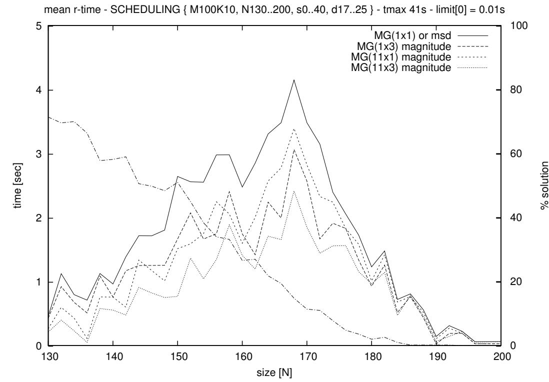

Fig. 3. Scheduling(MH vs. *msd*): left, frequency of failure to solve within *tmax*; right, mean r-time

QWH - In Fig 4, we again show robustness and run time, this time for balanced QWH(20). MH (10x3) again consistently outperforms min-size domain both in terms of robustness and run time. The graphs show two versions of MH, one with linear time-limit increase, and one with the order of magnitude increase every n restarts. All problems have solutions.

failure frequency [%]		
Size [N]	*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

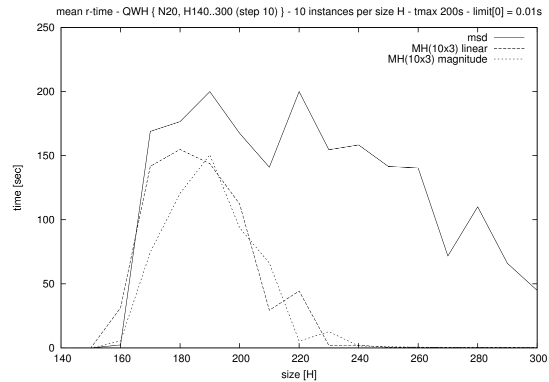

Fig. 4. QWH(MH vs. *msd*): left, frequency of failure to solve within *tmax*; right, mean r-time

5.2 Comparing to randomized-restarts on min domain

Randomized restarts (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. MH performed slightly better with time slices increased by a magnitude every loop of restarts, for which version we report the statistic on the frequency of failure.

failure frequency [%]		
Size [N]	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

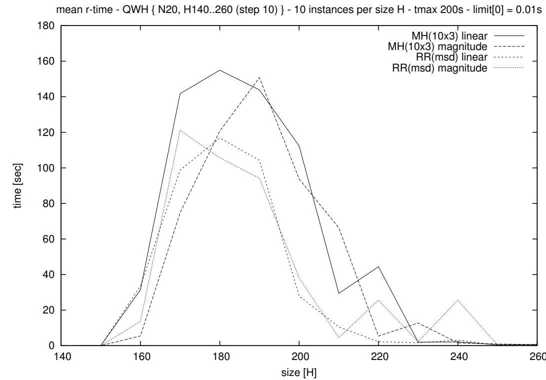

Fig. 5. QWH(MH vs. RR): left, frequency of failure to solve within *tmax*; right, mean r-time

Scheduling (Fig. 6) - MH clearly improves on RR at 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, i.e. increasing linearly every single restart (MH-Linear vs. RR-Linear), and increasing by an order of magnitude every loop (MH-Magnitude vs. RR-Magnitude). There is actually only a slight difference between the two 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

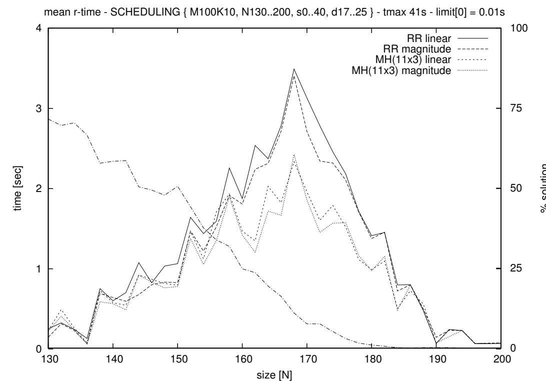

Fig. 6. Scheduling(MH vs. RR): left, frequency of failure to solve within *tmax*; right, mean r-time

6 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 degrading the 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.

Acknowledgements

This work is funded by Enterprise Ireland (SC/2003/81), with additional support from Science Foundation Ireland (00/PI.1/C075), and ILOG, SA.

References

1. D. Achlioptas, C. P. Gomes, H. Kautz, and B. Selman. Generating satisfiable problem instances. In *Proceeding of the Seventeenth National Conference on Artificial Intellicence (AAAI-00)*, pages 256–261, New Providence, RI: AAAI Press, 2000.
2. E. M. Arkin and E. B. Silverberg. Scheduling jobs with fixed start and end times. *Discrete Applied Mathematics*, 18:1–8, 1987.
3. J. C. Beck and M. S. Fox. Dynamic problem structure analysis as a basis for constraint-directed scheduling heuristics. *Artificial Intelligence*, 117(1):31–38, 2000.
4. P.; Beck, J.C.; Prosser and R.J. Wallace. Variable ordering heuristics show promise. In *Proceedings of the Tenth International Conference on Principles and Practice of Constraint Programming, CP'04*, pages 711–715, Montreal, Canada, 2004.
5. D. Frost and R. Dechter. Look-ahead value ordering for constraint satisfaction problems. In *Proceedings of the International Joint Conference on Artificial Intelligence, IJCAI'95*, pages 572–578, Montreal, Canada, 1995.
6. C. P. Gomes and B. Selman. Algorithm portfolios. *Artificial Intelligence*, 126(1-2):43–62, 2001.
7. C. P. Gomes and D. B. Shmoys. Approximations and randomization to boost csp techniques. *Annals of Operation Research*, 130:117–141, 2004.
8. T. Hulubei and B. O'Sullivan. Optimal refutations for constraint satisfaction problems. In *Proceedings of the International Joint Conference on Artificial Intelligence, IJCAI'05*, pages 163–168, Edimburgh, Scotland, 2005.