Borrett, J. & Tsang, E.P.K., Adaptive constraint satisfaction: the quickest first principle, in Mumford, C.L. & Lakhmi, J. (ed.), Computational Intelligence: Collaboration, Fusion and Emergence, Intelligent Systems Reference Library, Vol. 1, Springer, 2009, 203-230 (ISBN: 978-3-642-01798-8)

Adaptive Constraint Satisfaction: The Quickest First Principle

James E. Borrett and Edward P.K. Tsang

Abstract. The choice of a particular algorithm for solving a given class of constraint satisfaction problems is often confused by exceptional behaviour of algorithms. One method of reducing the impact of this exceptional behaviour is to adopt an adaptive philosophy to constraint satisfaction problem solving. In this report we describe one such adaptive algorithm, based on the principle of chaining. It is designed to avoid the phenomenon of exceptionally hard problem instances. Our algorithm shows how the speed of more naïve algorithms can be utilised safe in the knowledge that the exceptional behaviour can be bounded. Our work clearly demonstrates the potential benefits of the adaptive approach and opens a new front of research for the constraint satisfaction community.

1 Introduction

Constraint Satisfaction Problems occur in many areas of everyday life. These range from problems such as timetabling and transport planning to configuration problems and document layout design. In all cases, the notion of a Constraint Satisfaction Problem (CSP) is characterised by the need to assign values to elements of the problem instances, these values coming from a finite set of possibilities and subject to a set of rules or constraints [31] [23].

Once a CSP has been identified there are whole host of problem solving techniques which have been developed for solving them [23]. The most basic of these is the simple backtracking algorithm but more sophisticated algorithms such as look-ahead approaches have been shown to be highly effective [13] and are commonly used in commercial software libraries such as ILOG Solver [22]. Heuristic search has been applied to CSP with success, e.g. see [14] [18] [32], and have also been embedded in industrial packages such as iOpt [33].

James E. Borrett and Edward P.K. Tsang
University of Essex, Dept. of Computer Science, Wivenhoe Park, Colchester CO4 3SQ, United Kingdom
e-mail: {jborrett,edward}@essex.ac.uk

C.L. Mumford and L.C. Jain (Eds.): Computational Intelligence, ISRL 1, pp. 203–230.
springerlink.com

More formally, the constraint satisfaction problem (CSP) can be defined in terms of the triple <Z, D, C>, where Z is a set of variables, D is a mapping of the variables in Z to domains and C is a set of constraints [31]. Given this definition of a CSP, there are many ways in which different types of problem can be classified, in terms of the elements of Z, D and C[1]. This classification may then be used as a basis for the selection of a particular algorithm to solve that class of problems.

There is, however, a significant complication with the definition of CSP classes. Sometimes particular instances of problems in a class may exhibit exceptional qualities, in terms of the solving abilities of the chosen algorithm. One clear example of this is the phenomenon of exceptionally hard problem instances [28], or EHPs as they shall be referred to in this paper.

The example of EHPs is illustrative of the dilemma posed to the problem solver. There is a clear choice of either using a naïve algorithm which is likely to solve most instances very quickly, at the risk of catastrophic encounter with an EHP, or to choose a more complex algorithm, which has a far lower probability of encountering EHPs[2]. However, as is often the case, the use of more complex algorithms entails an overhead.[3]

One approach which can overcome this dilemma is to use a more flexible approach which we describe as adaptive constraint satisfaction. The notion of adaptive constraint satisfaction can be encapsulated in the following description:

> Adaptive Constraint Satisfaction is a general philosophy for solving constraint satisfaction problems. It aims to make use of the many algorithms and techniques available by relaxing the commitment to a single algorithm when solving a particular CSP, allowing for the active modification or switching of algorithms and models during the search process.

Built upon the Adaptive Constraint Satisfaction context was a set of research projects[4]. Adaptive constraint satisfaction is based on the belief that there is no "best algorithm" in constraint satisfaction – different algorithms work for different problem instances – an idea that was later articulated as the "No Free Lunch Theorem" [35, 36][5]. Based on this belief, Kwan et al [17] [30] developed a machine learning framework for learned mappings from CSPs to algorithms and heuristics. Given

[1] In [3] the issue of classifying different formulations of the same problem is considered.

[2] In the context of complete algorithms, [25], [26] suggest it is likely that investing in more complex algorithms, such as forward checking with conflict-directed backjumping [20], will decrease the frequency of encounters with EHPs.

[3] Given that EHPs are algorithm dependent, as explained above, another approach is to restart the search with, say a random algorithm. The difficulty in this approach is deciding when to restart. Abandoning the search prematurely means a waste of search effort; if one is not careful, one could end up restarting indefinitely. That motivated us to develop a mechanism to recognize thrashing.

[4] See Adaptive Constraint Satisfaction Project (1992-98) http://csp.bracil.net/acs.html

[5] According to this theorem "there is no free lunch when the probability distribution on problem instances is such that all problem solvers have identically distributed results". See Wikipedia http://en.wikipedia.org/wiki/No_free_lunch_in_search_and_optimization (accessed 18 August 2008)

a CSP, the algorithm picked may not work efficiently. This is because such mappings were generated statistically, which may not apply to every problem instance. The problem instance on hand may be "exceptionally hard" to the chosen algorithm and heuristic. Therefore, part of the Adaptive Constraint Satisfaction project was to develop measures for monitoring algorithms when they search. Every algorithm is designed to exploit certain characteristics of the problem instance. If an algorithm/heuristic does not do what it is supposed to do, it should be stopped, and a different algorithm/heuristic should be used. For example, lookahead algorithms [13] are designed to propagate constraints in order to prune the search space and detect dead-ends. If, during the search, it is found that not much of the search space is pruned, and a large amount of constraint propagation effort has resulted in few dead-ends being detected, the lookahead algorithm that is currently used should be replaced.

In this paper, we outline a particular instance of the adaptive approach where we make use of Algorithmic Chaining. The result is REBA (for Reduced Exceptional Behaviour Algorithm) which is designed to avoid the phenomenon of exceptionally hard problems in the so called easy region for solvable CSPs. REBA operates on complete search methods – methods that explore the search space systematically and entirely if necessary.

2 The Adaptive Strategy

We have defined adaptive constraint satisfaction as a general approach to solving CSPs. Within that approach there are many possible strategies. We examine one particular adaptive strategy, designed to reduce the significance of EHPs by utilising algorithmic chaining. Algorithmic chaining uses a set of algorithms, arranged in a pre-determined order, combined with a switching mechanism. The switching mechanism monitors the search process of the current algorithm and, should certain conditions occur, stops the current algorithm, trying again with the next algorithm in the chain. In this section we discuss these two elements of the strategy.

2.1 Chain Design

As noted in [25], [26] the phenomenon of EHPs appears to affect different algorithms to different degrees. However, the trend tends to be for more naïve algorithms, such as simple chronological backtracking algorithms, to be more susceptible. This presents us with two potentially useful measures for ranking algorithms. The first is the cost to solve 'normal' occurrences of CSPs (measured by the median cost), and the second is the algorithms sensitivity to EHPs. An example of possible differences in ranking is given in Table 1.

If we can determine similar rankings to those in Table 1, we would have enough information to design a useful chain for solving CSPs in the easy region whilst increasing the likelihood of avoiding the potentially catastrophic effects of

Table 1 Example showing how the ranking of algorithms can differ when based on median cost of solving CSPs, and sensitivity to EHPs

Rank	Algorithm Complexity	Median Cost	Sensitivity to EHPs
1	X	X	Z
2	Y	Y	Y
3	Z	Z	X

encountering an EHP. The chain can simply be set to an ordering based on the "Quickest First Principle" (QFP), where quickest indicates the algorithm with the best median performance.

We wanted to design an algorithm for solving easy solvable problems without failing in EHPs. Using QFP means that we always have the potential for solving the CSPs quickly. However, if we can detect that the current algorithm is not working well, we could switch to the next quickest algorithm, and so on. As a result we can still benefit from the speed of the naïve algorithms while at the same time having the capability to resort to more complex algorithms in the event that a switch scenario is detected.

While there is some overhead involved in this approach, the benefits can be considerable. For example, the ability to use a simple algorithm can result in an order of magnitude gain in performance over its more complex counterparts. Another advantage is that in the event of a bad initial choice of algorithm, we are not stuck with it. Mistakes of this nature will be rectified when we switch away from the bad choice.

2.2 Switching Policy

The main requirements of the switching mechanism are that it can detect the phenomena you wish to avoid, while adding only minimal overheads to the basic algorithm. For REBA this means we need to predict the thrashing type behaviour associated with EHPs encountered by naïve algorithms, using a simple and efficient prediction method.

There appear to be many types of thrashing in CSPs. [25], [26] note the basic thrashing scenario is often seen in chronological backtracking algorithms such as forward checking [13]. This is the worst type of thrashing, where the algorithm visits all nodes in a sub-tree of the search space when it is futile to do so. It is not experienced by more complex algorithms, such as intelligent backjumping algorithms. However the idea of a search sub-space being repeatedly visited when it is futile to do so still occurs in these algorithms, the main difference being the amount of the sub-space visited.

At the heart of the switching mechanism of REBA is the Monitor Search Level (MSL) thrashing predictor which is described in detail in Section 3.2. MSL represents one possible mechanism which attempts to predict when thrashing type behaviour is likely to occur such that only a small portion of any futile sub search space is actually explored by the algorithm in question. Using a sensitivity

threshold supplied to it, the predictor will suggest that a switch is necessary if the threshold is reached.

3 The Reduced Exceptional Behaviour Algorithm (REBA)

Having outlined the basic strategy for our Reduced Exceptional Behaviour Algorithm, we give more details of its design. We also give a description of the prediction mechanism used by REBA.

3.1 The REBA Algorithm Chain

The chain used by REBA is designed using the principles outlined in Section 2. This chain uses a selection of algorithms with good median performance on easy soluble CSPs, and a selection of algorithms with good worst case performance. These cover a range of complete search techniques including features such as forward checking, backjumping and heuristics which cover both static and dynamic variable orderings. No stochastic algorithms are considered for REBA, but this should not rule out the possibility of using them in alternative adaptive approaches. Space would not allow us to go into details of these algorithms. Relevant pointers are provided here. [31] explains most of these algorithms. In a way, it is not essential to understand details of these algorithms. For this paper, the relevant point is that they cover a wide range of algorithms and heuristics with diversified strength.

Having carried out some preliminary investigations, we chose to use the following algorithms;

BM+MWO	back-marking [9] with the minimum width ordering heuristic [6]
BMCBJ+MWO	back-marking with conflict-directed backjumping [20] with the minimum width ordering heuristic
BMCBJ+MDO	back-marking with conflict-directed backjumping [20] with the maximum degree ordering heuristic[6]
FCCBJ+BZ	forward checking with conflict-directed backjumping [20] with the Brélaz ordering heuristic [29], [4]
MAC+MDO	Maintain Arc Consistency [24] with the maximum degree ordering heuristic

We propose to use these algorithms in the following chain to tackle problem instances in the easy, soluble region:

BM+MWO → BMCBJ+MWO → BMCBJ+MDO → FCCBJ+BZ → MAC+MDO

The reasoning behind this chain is that BM+MWO is very fast for many easy soluble problem instances, but very susceptible to EHPs. However, it might succeed in a very quick solution, otherwise thrashing will be detected. In the event that

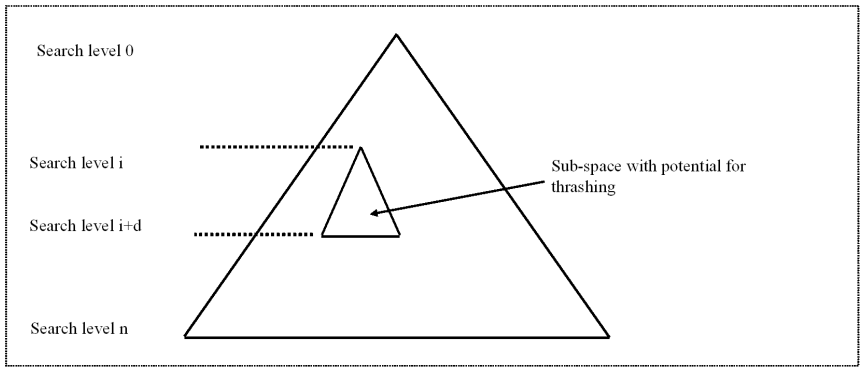


Fig. 1 An example of a sub search space

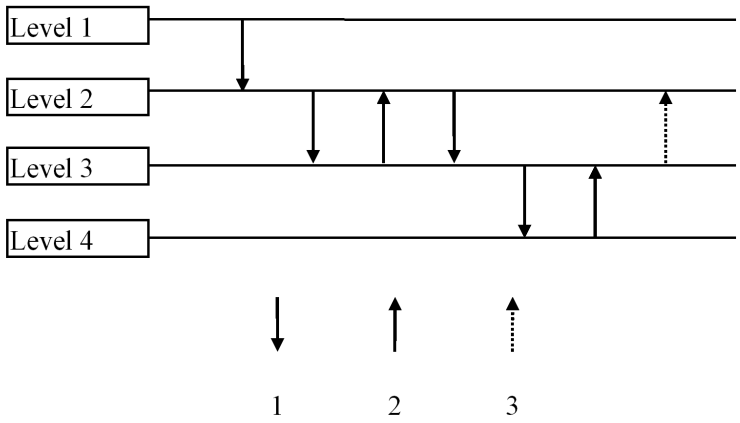


Fig. 2 The types of progress during search (see text for explanations)

BM+MWO fails, we try adding intelligent backjumping to it. If this fails, we try changing the ordering, since a bad ordering is often a contributing factor to EHPs [28]. If these simpler algorithms fall victim to an EHP, we attempt to use a form of forward checking with conflict-directed backjumping and a dynamic variable ordering. Finally, if this fails, we resort to another algorithm which has relatively low susceptibility to EHPs, MAC+MDO.

3.2 The Monitor Search Level (MSL) Thrashing Predictor

In this section we describe the Monitor Search Level (MSL) thrashing predictor. We describe the behaviour MSL watches for, and explain how it decides when this behaviour is sufficiently clear for thrashing to be predicted.

As a basis for the design of MSL we defined the following functional specification;

> Given a CSP, an algorithm, and a variable ordering, the predictor should monitor the progress of the search and be able to predict if thrashing is likely to occur during the search.

One indication of thrashing is when the search from a particular level i never proceeds beyond a certain depth, d, and that a large proportion of the search space between level i and level $i+d$ is explored (i.e. little pruning takes place between these two levels, see Figure 1). Such a situation can occur when the culprits of the failure at level $i+d$ precede the level i. MSL is a simple method which uses this observation to predict thrashing type behaviour.

Before discussing MSL in more detail, we must identify three distinct types of progress which occur during search. These are presented in figure 2.The types of progress are defined as;

1. A value is found for the current variable which is compatible with all previous assignments, or future variables in the case of lookahead algorithms. For example the second arrow in Figure 2, where a value is found for the variable at level 2 which is compatible with the value assigned to the variable at level 1.
2. Backtracking occurs after finding no values for the current variable which are compatible with previous assignments, or future variables in the case of lookahead algorithms. For example the third arrow in Figure 2, where no value can be found for the variable at level 3 which is compatible with the current assignments of the variables at levels 1 and 2. This will be known as a No Assigned Value (NAV) backtrack. The NAV backtrack occurs at the tail of the arrow, level 3. At the head of the arrow, level 2 learns of an Unsuccessful Subspace Search (USS).
3. Backtracking occurs, but only after at least one value has been found for the current variable which is compatible with the assignments of previous variables, or future variables in the case of lookahead algorithms (Meaning the search must have progressed at least one level further down than the current one). For example the seventh arrow in Figure 2, where a value for the variable at level 3 has been found which is compatible with the assignments of the variables at levels 1 and 2, but is later rejected because no value can be found for the variable at level 4. This will be known as a Successfully Assigned Values (SAV) backtrack. The SAV backtrack occurs at the tail of the arrow, level 3. At the head of the arrow, level 2 learns of a USS.

During the search MSL keeps track of the last level at which a NAV backtrack occurred. This is considered to be the deepest level of the current search sub-space. We will refer to this level as DEEPEST.

In addition, for each level in the search, MSL keeps track of two values. Firstly a count indicating the number of USS's which returned to the level with the same value for DEEPEST. Secondly a record of the value of DEEPEST when this count

Table 2 Possible actions of MSL on $count_i$ and DL_i for level i

	(1) DEEPEST $< DL_i$	(2) DEEPEST $= DL_i$	(3) DEEPEST $> DL_i$
(a) USS	No action	Increase $count_i$ by 1; Check count against threshold	Set $count_i$ to 1; Set DL_i to DEEPEST
(b) NAV Backtrack	Set DEEPEST to i	Set DEEPEST to i	Not Possible
(c) SAV Backtrack	Reset $count_i$ to 0; Set DL_i to DEEPEST	No action	Not Possible

is started. We will refer to these values as $count_i$ and DL_i respectively, where i is the level they refer to.

In considering how the count is maintained, we must examine the seven possible cases. These depend on whether a USS, a NAV backtrack or a SAV backtrack is occurring, and what the value of DEEPEST is compared to the value of DL_i for the level. Table 2 illustrates the different actions taken at a given level, i, depending on these circumstances.

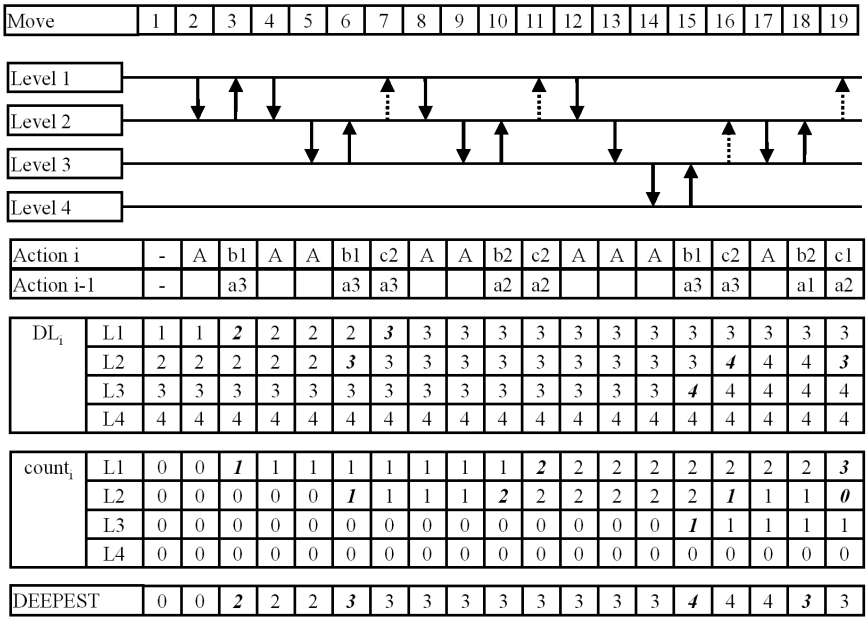
Some points should be noted here:

- DEEPEST and $count_i$ are initialised to 0 and DL_i are initialised to i
- DEEPEST can only be changed by a NAV backtrack occurring, and always changes when such a backtrack occurs.

Figure 3 gives an example illustrating the possible situations encountered by MSL. Each column in Figure 3 represents either an assignment, a NAV backtrack, or a SAV backtrack together with a USS if applicable (with the exception of the first column). The numbers below the arrow indicate the values of DL_1,..,DL_4, $count_1$,...,$count_4$ and DEEPEST **after** the actions for that column have been carried out. The values of the actions indicate which entries in Table 2 apply to the above arrow[7]. This includes actions at both the tail and the head of the arrow. The first column simply shows the initial values before the search begins.

As an example consider columns 14 to 16. Column 14 shows a simple assignment to the variable at level 3, action A. No further actions take place. Column 15 then shows a NAV backtrack from the variable at level 4. When the backtrack occurs, $DL_4 = 4$ and DEEPEST = 3, so $DL_4 >$ DEEPEST and entry $b1$ in Table 2 applies to level 4. As a result DEEPEST is set to the value of i, i.e. DEEPEST = 4. At the

[7] The entry A indicates a successful assignment, no action is taken.



Move	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Action i	-	A	b1	A	A	b1	c2	A	A	b2	c2	A	A	A	b1	c2	A	b2	c1
Action i-1	-		a3			a3	a3			a2	a2				a3	a3		a1	a2

DL_i																			
	L1	1	1	***2***	2	2	2	***3***	3	3	3	3	3	3	3	3	3	3	3
	L2	2	2	2	2	2	***3***	3	3	3	3	3	3	3	3	***4***	4	4	***3***
	L3	3	3	3	3	3	3	3	3	3	3	3	3	3	***4***	4	4	4	4
	L4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4

$count_i$																			
	L1	0	0	***1***	1	1	1	1	1	1	1	***2***	2	2	2	2	2	2	***3***
	L2	0	0	0	0	0	***1***	1	1	1	***2***	2	2	2	2	2	***1***	1	1	***0***
	L3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	***1***	1	1	1	1
	L4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

DEEPEST	0	0	***2***	2	2	***3***	3	3	3	3	3	3	3	3	***4***	4	4	***3***	3

Fig. 3 Example search

head of the arrow USS entry *a*3 applies (because DEEPEST = 4 and $DL_3 = 3$) and $count_3$ is set to 1 with DL_3 being set to DEEPEST.

Column 16 shows a SAV backtrack from the variable at level 3. When the backtrack occurs, $DL_3 = 4$ and DEEPEST = 4. Since DL_3 = DEEPEST entry *c*2 in Table 2 applies and no action is taken at level 3. At the head of the arrow USS entry *a*3 applies and $count_2$ is set to 1 with DL_2 being set to DEEPEST.

3.2.1 Effectiveness of Thrashing Prediction Mechanisms

Having defined the function of our prediction mechanism, we also define a set of criteria for evaluating its effectiveness. These criteria are based on three main functions;

i It should predict as exceptionally hard those problem instances with high search cost for the current algorithm.
ii The computational cost of predicting a CSP to be exceptionally hard should be low and preferably not exceed the median cost. It should also be cheap in terms of space.
iii It should not be so sensitive that too many problem instances are predicted to be exceptionally hard. A high proportion of the problem instances with search

costs of median or lower should not be predicted to be exceptionally hard for the current algorithm.

3.3 The REBA Switching Mechanism

The MSL predictor is used by REBA for its switching mechanism. This is done by REBA supplying the predictor with a formula for calculating the threshold. If the threshold is exceeded, then MSL suggests that a switch should take place. As a result, REBA will switch to the next algorithm in the chain.

We have experimented with a threshold based on the domain size of the variables, and the number of levels separating the current level i and DL_i. The base threshold is a multiple of the domain size. The number of separating levels is taken as DL_i - i. The more separating levels, the lower the threshold has to be for switching to occur. The formula used is;

$$Threshold = base * \left(\frac{n - separation}{n} \right)$$

where: - base is the base threshold, which is a linear function of the domain size
- n is the number of variables,
- $separation$ is the number of separating levels (DL_i- i).

The threshold is adjusted according to $separation$ to improve the sensitivity of detection when the subspace is only searched sparsely, as might be the case with intelligent backjumping algorithms.

Note that in subsequent experiments a suffix is given to the name of REBA. This suffix indicates the multiples of the domain size used for the base threshold.

4 Experiments

In order to evaluate the overall performance of REBA and the effectiveness of its switching mechanism we carried out an experiment on different classes of easy soluble CSPs (which is what REBA is designed to tackle). This section describes details of our experiment as well as presenting our results.

4.1 Experimental Design

The main aim of our experiment was to compare the performance of REBA with two types of algorithms - those exhibiting good median performance in the easy soluble region, and those that have a good worst case performance on easy soluble region. Randomly generated CSPs are used to evaluate REBA. They allow us to control the tightness of problem classes, and therefore select appropriate problem classes for experimentation.

The actual CSPs we used were based on randomly generated binary CSPs classified by the tuple <n, m, p1, p2>, where the elements of the tuple are defined as;

- n number of variables
- m uniform domain size
- $p1$ density of constraints in the constraint graph
- $p2$ tightness of individual constraints[8] i.e. the percentage of incompatible assignments between the two variables involved in the constraint

Specifically, we wanted to conduct our experiments on problems in the so-called easy soluble region where exceptionally hard problem instances were likely to occur. As a result, we chose the class <50, 10 , 0.1, 0.35 - 0.5 >. This range of $p2$ gives us a spread of problem instances in the region of interest and it also includes some of the sets of problems used in [25] and [26], where EHPs were investigated.

The algorithms we chose for comparison, based on initial tests of problem instances in the class description above, were as follows;

BMCBJ+MWO	back-marking with conflict-directed backjumping with the static minimum width ordering - this combination gives a low median performance but has a sensitive worst case performance in the region of interest.
FCCBJ+BZ	forward checking with conflict-directed backjumping with the dynamic Brélaz ordering - this combination gives a relatively high median performance but a good worst case performance in the region of interest.
MAC+MDO	maintain arc-consistency with the static maximum degree ordering - this combination also gives a relatively high median performance but a good worst case performance in the region of interest.

The CSPs for our experiments were generated at intervals of $p2$ of 0.01 and the sample size for each data point was 1000. In order to limit the impact of EHPs on our experimentation time, we limited the actual process CPU time for any given run to 30 minutes. Where this time is exceeded, the compatibility check count up to that time was recorded[9].

The effect of using such a limit is that for a few data points, for the BMCBJ+MWO combination, the limit was reached. This does not detract from the essence of our results, however, since the effect of any EHP is still clearly visible. The truncated values are many orders of magnitude above the median search cost.

[9] Note that the algorithms were implemented in C++ and run on DEC Alpha 3000 Model 600 AXP workstations running at 175 MHz.

4.2 The Effectiveness of REBA

The results of our experiment in measuring the effectiveness of REBA are presented in Figures 4–7[10]. The results clearly show that the use of algorithmic chaining in REBA has produced a good worst case performance where the impact of EHPs has been significantly reduced. This is evident in the worst case plots of Figures 5 and 7. REBA even outperforms FCCBJ+BZ in many cases. At the same time, the median

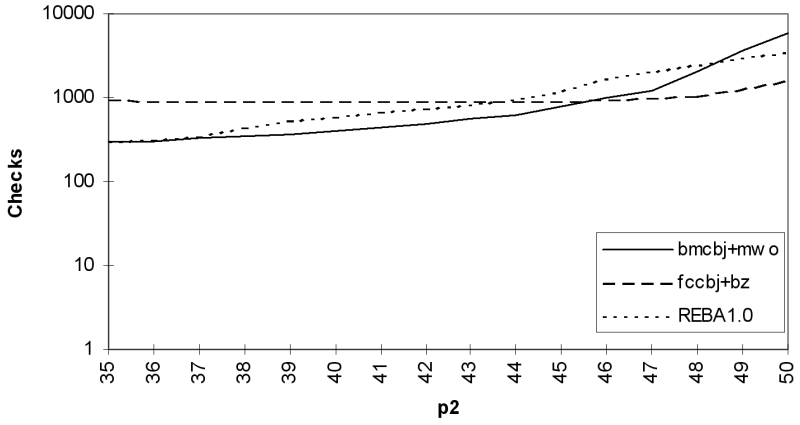


Fig. 4 Median performance on 50 variable problems in terms of compatibility checks

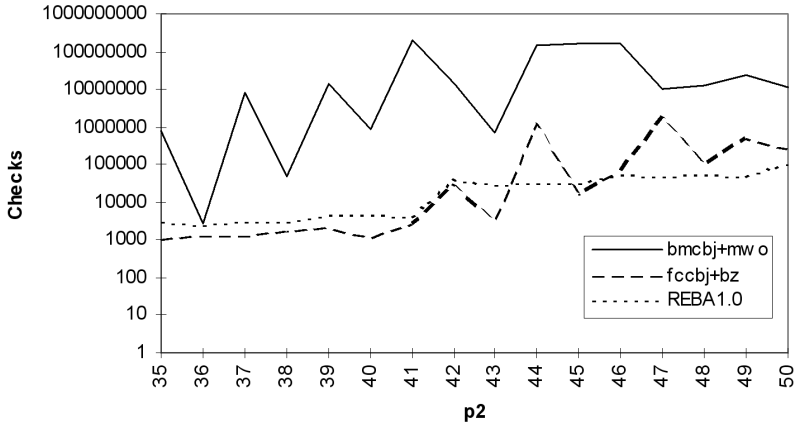


Fig. 5 Worst case performance on 50 variable problems in terms of compatibility checks

[10] We only present cpu time results for MAC since our implementation is the same as that of [24] where the compatibility check count is not a true reflection of the work done by MAC.

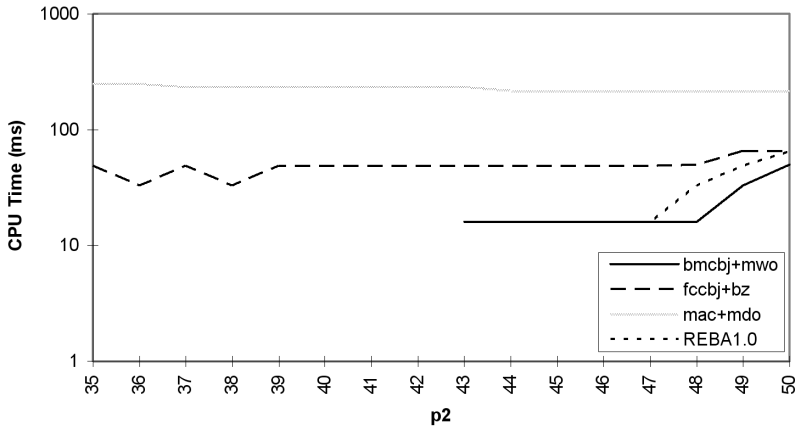


Fig. 6 Median performance on 50 variable problems in terms of cpu time. (Note that where the plot for REBA and BMCBJ+MWO does not exist this means the median time was less that one clock cycle and hence does not show in the logarithmic scale)

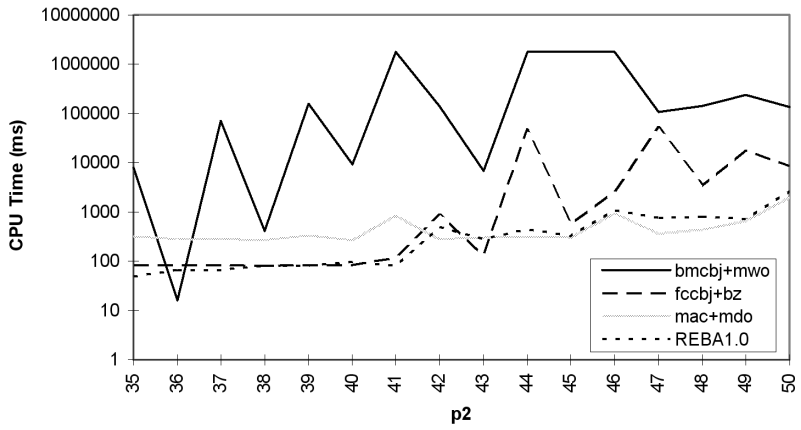


Fig. 7 Worst case performance on 50 variable problems in terms of cpu time

performance of REBA is much better than that of the more complex algorithms, in most cases. This is particularly apparent when the CPU time is considered as in Figures 6 and 7.

It should be noted that we have tested REBA on problems in the easy region. This is because we advocate that different types of problem would be tackled by different algorithms as noted in [30]. REBA, by design, appears to be useful in tackling problems in the easy region on the soluble side of the phase transition. It is the subject of further work to investigate the applicability of the strategies used in REBA to tackling other problem types such as those in the phase transition.

4.3 Evaluation of the MSL Predictor

To see how effective the switch detection mechanism in REBA is, we carried out a further experiment. This time, we did not run the chain of algorithms. Instead, we ran a version of BM+MWO, which included the MSL predictor, and monitored where a switch was predicted (if one was required). If a switch was predicted, the number of compatibility checks was recorded and the algorithm was allowed to continue running to completion to see what the actual outcome would have been[11]. We also repeated this experiment for an intelligent backjumping algorithm, BM-CBJ+MWO, allowing us to observe the effectiveness of MSL in these two types of algorithm.

For the BM+MWO combination, a problem set of 1000 CSPs were generated with the specification $<50, 10, 0.1, 0.4>$. For the BMCBJ+MWO combination 1000 CSPs with the specification $<50, 10, 0.1, 0.5>$. This difference in $p2$ is a reflection of the location where REBA was observed to have switched from these algorithms in the experiment detailed in Section 4.2.

In Section 3 we defined three criteria for a evaluating a thrashing prediction mechanism. We present our results in three ways to address these criteria. In Figures 8 and 9 we see how effective MSL is at filtering out problems where the actual cost of search to completion would have been high, including the possibility of EHPs. These histograms show the actual cost to completion of all the instances where a switch would have taken place[12](of which there were 589 for BM+MWO and 693 for BMCBJ+MWO).

These two figures show how there are many high cost searches predicted by MSL to be thrashing.

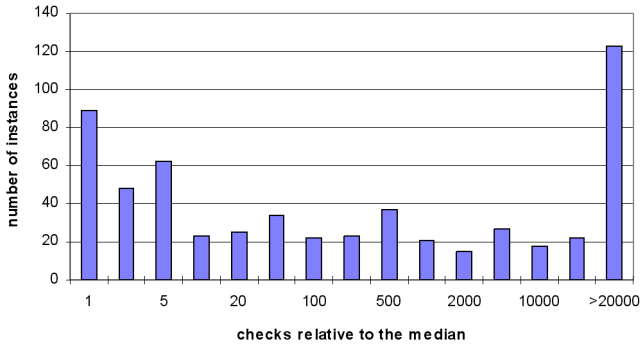


Fig. 8 Ultimate search cost for BM+MWO had a switch not been predicted (total of 589 instances)

[11] For the purposes of this experiment we used a base threshold equal to the domain size of the variables.

[12] The results are presented as multiples of the median search cost when considering the cost to completion for all CSPs in the sample of 1000.

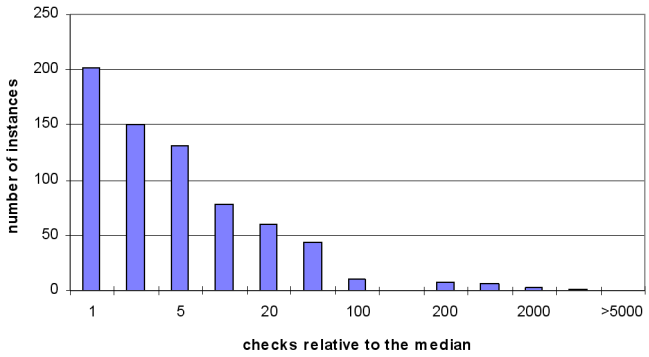


Fig. 9 Ultimate search cost for BMCBJ+MWO had a switch not been predicted (total of 693 instances)

The second criterion was that the cost to detection should be low. Figures 10 and 11 show the actual search cost up to detection for the instances where a switch was suggested.

As can be seen from these figures the performance is good, since the median cost for predicting a switch in BM+MWO was always less than the median search cost when all CSPs are considered. For BMCBJ+MWO a similar result can be seen, with the exception of a few cases. However, even with these exceptions, there are no cases where the cost exceeds five times the overall median.

Finally, the third criterion was that the prediction mechanism should not be too sensitive and prevent completion of search for the many problem instances that would have only had median cost to solve to completion. Figures 12 and 13 show the cost of search for all the problem instances where no switch was predicted place (of which there were 411 for BM+MWO and 307 for BMCBJ+MWO).

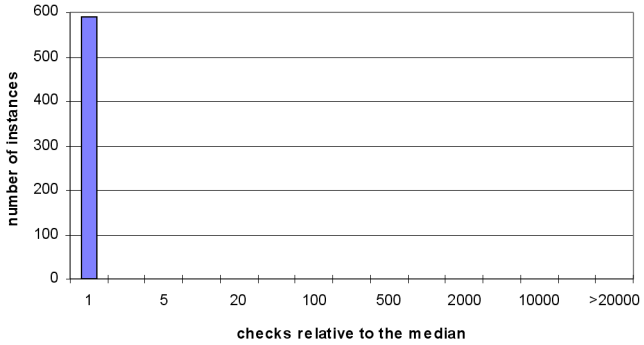


Fig. 10 Cost to predict a switch for BM+MWO (total of 589 instances)

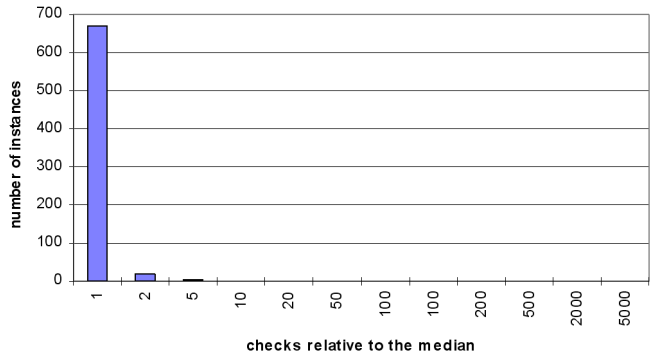


Fig. 11 Cost to predict a switch for BMCBJ+MWO (total of 693 instances)

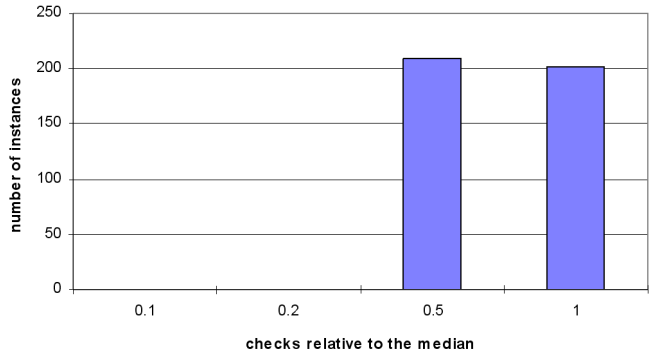


Fig. 12 Search cost for problems where no switch was predicted for BM+MWO (total of 411 instances)

This clearly shows that no high cost problem instances are allowed through and that there were many low cost problems let through. For BM+MWO, the maximum search cost for a CSP in this set was less than the median for all problems. In the case of BMCBJ+MWO, the maximum never exceeds five times the median.

From Figures 9–14 it is clear that the MSL predictor used for this version of REBA, with a base threshold of 1.0, has performed very effectively, and that the criteria laid out in Section 3.2 are largely fulfilled.

There is obviously a trade off when choosing the value for the threshold such that no exceptionally hard problems are encountered, whilst at the same time allowing the majority of the easier problems to be solved. The base threshold we have used was equal to the domain size of the variables and was the same for all algorithms. However, it may be possible to improve the effectiveness of algorithms such as REBA by using a different threshold, or perhaps by using different thresholds for the different algorithms in the chain.

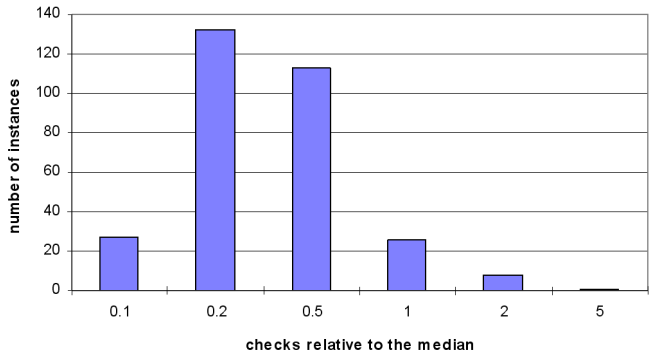


Fig. 13 Search cost for problems where no switch was predicted for BMCBJ+MWO (total of 307 instances)

We have experimented with different thresholds and find that they also produce good results when compared to the algorithms used in the above tests. We have also looked at how REBA performs with larger problem sizes. Again, REBA performs well. These results are given in Section A2.

5 Discussion

In this chapter we have demonstrated the potential of adaptive constraint satisfaction. We have outlined a particular application of the adaptive approach using the technique known as algorithmic chaining. This technique was incorporated in an algorithm that we have named REBA, and has been shown to be effective in reducing susceptibility to exceptionally hard problem instances.

The REBA algorithm makes use of a mechanism for predicting when thrashing type behaviour is likely to occur. This notion of prediction is one of the keys to the adaptive approach since it is prediction that allows algorithms to avoid search spaces before they can impact significantly on the overall search. The MSL mechanism used here is computationally very cheap and it has been shown to be reasonably accurate.

Experiments with the REBA algorithm, which is specifically designed to reduce the impact of exceptionally hard problem instances, show that it is possible to take advantage of the speed of basic constraint satisfaction algorithms when solving easy, soluble problem instances, while at the same time allowing us to bound the exceptional behaviour of these algorithms when exceptional problem instances are encountered. The principle of using the quickest algorithm first means that the best case performance of the naïve algorithms always has a chance of being achieved. It also gives the opportunity for fast solutions to be provided in the event that "exceptionally easy" problem instances are encountered - this could be significant if a similar method were to be used on, for example, hard classes of CSPs.

REBA represents a novel approach which demonstrates the potential of collaboration between algorithms. Exceptionally hard problem instances are so punitive in terms of cost that effective detection of potential traps for naïve algorithms using a low cost detection mechanism such as MSL that it is possible to make use of more sophisticated algorithms when they are needed, without incurring their general overhead.

REBA demonstrates a number of key ideas of adaptive constraint satisfaction. First, it recognizes that no algorithm is best for every CSP, as the No Free Lunch Theorem does. It also demonstrates that search performance could be monitored to see whether the algorithm is achieving what it is designed to achieve. (In fact, there is no reason why algorithms should not be monitored beyond the constraint satisfaction context.) REBA also demonstrates that efficiency can be gained by a rigid chain of algorithms.

Gomes et al [12] studied when expensive search happens in a given algorithm, which is highly related to thrashing detection in REBA. Dynamic restart was also investigated by a number of other works. Kautz et al [38] defined a set of policies for restarting the search. Gagliolo and Schmidhuber [8, 7] proposed to model the runtime distribution – if training is possible – and use the estimated runtime distribution to decide when to restart. In the Solution-Guided Search algorithm, Beck [2] set, before the search starts, limits on the number of fails that the search is allowed to encounter. One could imagine using REBA's thrashing predictor to dynamically set this limit.

Using a portfolio of algorithms for constraint satisfaction has gained momentum in the last decade, see, for example, [15], [11] and [39]. Once a portfolio of algorithms is involved, selecting the right algorithm for the job becomes part of the research agenda in [10] and [39].

Many attempts have been made to learn from the problem solving experience. The idea of selecting the right heuristic algorithm during run time was developed by Allen and Minton [1]. Epstein et. al. [5] sought to learn search order heuristics during problem solving. Related to these ideas, Minton [37] demonstrated the possibility of synthesizing heuristics. Kern (2005) used population-based incremental learning to select algorithms and parameters. Kern's work is embedded in iOpt [33], which is used in many real-life dynamic problems, such as service scheduling [34].

This piece of work has opened many new areas of future work. One could further investigate the use of chains and similar methods of choosing appropriate algorithms to switch to in types of problems other than soluble easy CSPs. One could also look at other methods for detecting when it would be useful to switch between algorithms. This would involve identifying useful information that can be gathered during search. The actual process of switching could also be a source of improvement in efficiency. Ideally, information collected during the search could be used for selecting the new algorithm or heuristic. When switches take place, information gathered so far could be transferred to successive algorithms.

Acknowledgements. This work was supported by EPSRC research grant ref. GR/J/42878. The authors would like to thank Alvin Kwan for his useful comments on the contents of

this paper. Natasha Walsh participated in the early part of this research. Christine Mumford (editor) and the anonymous referee provided us with insightful comments, which helped to improve the quality of this paper.

Please note: this paper is based on an extended form of Borrett, J., Tsang, E.P.K. & Walsh, N.R., Adaptive constraint satisfaction: the quickest first principle, Proceedings, 12th European Conference on AI, Budapest, Hungary, 1996, p.160-164.

Appendix

A.1 Tables of results for Figures 4–7

Table 3 Data for Figure 4, median performance on 50 variable problems in terms of compatibility checks

p2	bmcbj+mwo	fccbj+bz	REBA1.0
35	296	934	300
36	308	929	319
37	324.5	920	344
38	342	914	446
39	367	907.5	530
40	399.5	904	601
41	435.5	899	671
42	489.5	900	742.5
43	575	897	840
44	620	906	932
45	799	915	1216.5
46	1021.5	932	1691
47	1226	982	2037
48	2090	1064	2475.5
49	3624.5	1244	3002
50	5785	1628	3491.5

Table 4 Data for Figure 5, worst case performance on 50 variable problems in terms of compatibility checks

p2	bmcbj+mwo	fccbj+bz	REBA1.0
35	815674	1005	3044
36	2639	1274	2342
37	8067955	1320	2886
38	50716	1828	3092
39	13907031	2103	4787
40	913249	1139	4601
41	2E+08	2737	3997
42	14676577	29868	37618
43	698687	3071	27071
44	1.43E+08	1242863	32790
45	1.57E+08	16877	30992
46	1.56E+08	66307	53962
47	9738619	1875137	48700
48	12706257	107156	52169
49	23113988	524932	50650
50	11733913	269627	100697

Table 5 Data for Figure 6, median performance on 50 variable problems in terms of cpu time

p2	bmcbj+mwo	fccbj+bz	mac+mdo	REBA1.0
35	0	49	250	0
36	0	33	249	0
37	0	49	233	0
38	0	33	233	0
39	0	49	233	0
40	0	49	233	0
41	0	49	233	0
42	0	49	233	0
43	16	49	233	0
44	16	49	216	16
45	16	49	216	16
46	16	49	216	16
47	16	49	216	16
48	16	50	216	33
49	33	66	216	49
50	50	66	216	65

Table 6 Data for Figure 7, worst case performance on 50 variable problems in terms of cpu time

p2	bmcbj+mwo	fccbj+bz	mac+mdo	REBA1.0
35	8016	83	316	49
36	16	83	283	66
37	71233	83	283	66
38	416	82	266	82
39	157783	83	333	82
40	9283	83	266	99
41	1800000	116	850	82
42	138433	916	283	498
43	6916	132	300	283
44	1800000	50266	316	448
45	1800000	583	300	332
46	1800000	2582	950	1082
47	107600	55149	366	766
48	144550	3549	433	815
49	237282	17632	650	732
50	135583	8549	1983	2615

A.2 Results for 100 Variables

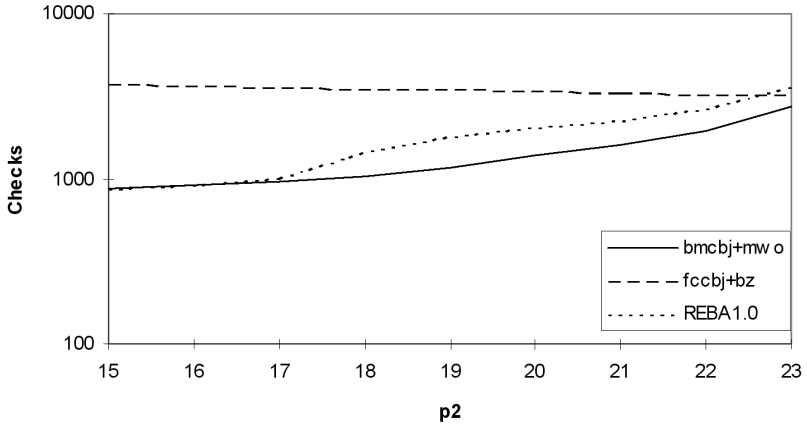


Fig. 14 Median performance on 100 variable problems in terms of compatibility checks

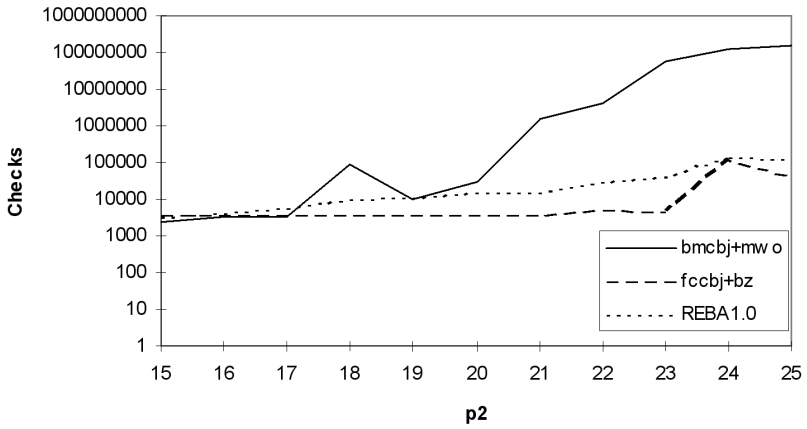


Fig. 15 Worst case performance on 100 variable problems in terms of compatibility checks

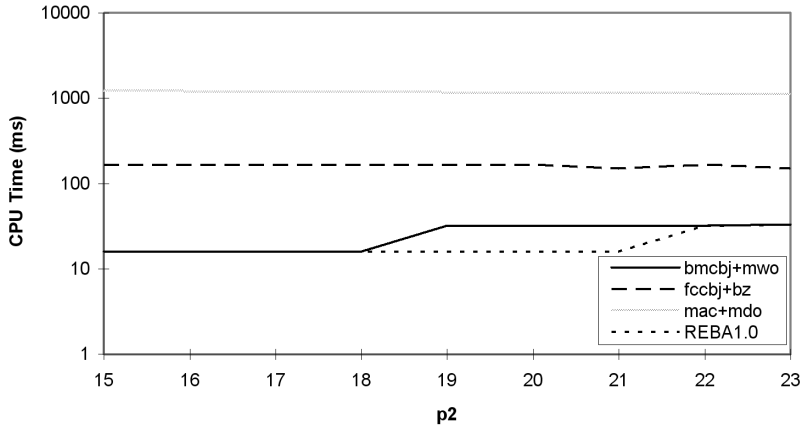


Fig. 16 Median performance on 100 variable problems in terms of cpu time

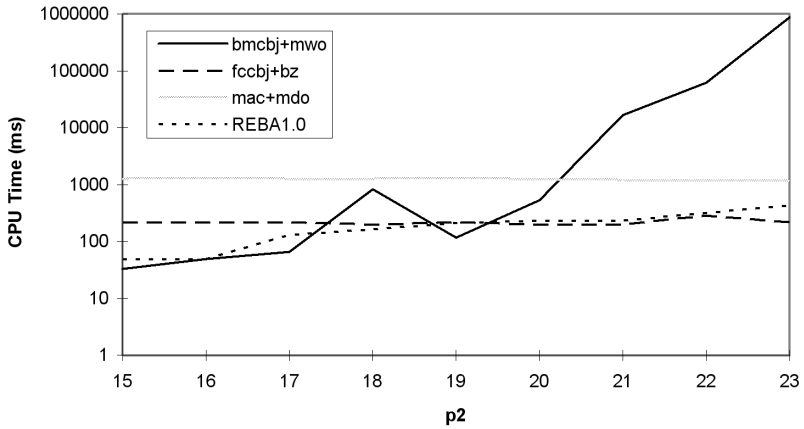


Fig. 17 Worst case performance on 100 variable problems in terms of cpu time

A.3 Tables of results for Figures 14 – 17

Table 7 Data for Figure 14, median performance on 100 variable problems in terms of compatibility checks

p2	bmcbj+mwo	fccbj+bz	REBA1.0
15	865	3757	866
16	918	3682	928
17	971	3607	1010.5
18	1047	3533	1446.5
19	1177.5	3465	1830
20	1383.5	3396	2037.5
21	1598.5	3335	2275.5
22	1940.5	3271	2694
23	2756	3222.5	3606

Table 8 Data for Figure 15, worst case performance on 100 variable problems in terms of compatibility checks

p2	bmcbj+mwo	fccbj+bz	REBA1.0
15	2353	3920	3261
16	3387	3834	4100
17	3527	3773	5996
18	90854	3706	10388
19	10129	3623	11008
20	30716	3584	16016
21	1527678	3804	16256
22	4266115	5348	29393
23	59600500	4400	41197

Table 9 Data for Figure 16, median performance on 100 variable problems in terms of cpu time

p2	bmcbj+mwo	fccbj+bz	mac+mdo	REBA1.0
15	16	166	1232	0
16	16	166	1216	0
17	16	166	1200	16
18	16	166	1199	16
19	32	166	1183	16
20	32	166	1166	16
21	32	150	1166	16
22	32	166	1150	32
23	33	150	1133	33

Table 10 Data for Figure 17, worst case performance on 100 variable problems in terms of cpu time

p2	bmcbj+mwo	fccbj+bz	mac+mdo	REBA1.0
15	33	216	1266	49
16	49	216	1283	49
17	66	216	1266	132
18	832	200	1266	164
19	116	216	1333	215
20	532	200	1249	232
21	16800	200	1216	232
22	61533	283	1199	315
23	884032	216	1182	432

A.4 REBA Results for Different Base Thresholds

Table 11 REBA results for base thresholds of 1.5 and 2.0 for CSPs used in Figures 3–6

p2	median checks		median cpu time		worst case checks		worst case cpu time	
	REBA2.0	REBA1.5	REBA2.0	REBA1.5	REBA2.0	REBA1.5	REBA2.0	REBA1.5
35	299	299	0	0	2068	2057	33	33
36	314	314	0	0	4016	3111	98	49
37	339	339.5	0	0	3421	3143	48	49
38	375.5	380.5	0	0	6000	4709	66	49
39	489.5	520.5	0	0	6356	5837	99	50
40	608.5	602	0	0	6387	6387	82	98
41	716	698	0	0	6567	6028	83	98
42	782.5	762	0	0	8938	8437	98	82
43	895.5	865	0	0	14045	13285	198	116
44	969.5	944	16	0	24296	17351	298	183
45	1181.5	1158	16	16	27050	16279	316	199
46	1461	1408.5	16	16	49321	34802	832	750
47	1730	1744.5	16	16	101768	98090	2482	2533
48	2834	2766	32	32	56755	45932	1098	848
49	4407	4026.5	49	49	133510	146291	2799	3482
50	5962	5172.5	66	66	240225	221734	5682	5132

Table 12 REBA results for base thresholds of 1.5 and 2.0 for CSPs used in Figures 14–17

p2	median checks		median cpu time		worst case checks		worst case cpu time	
	REBA2.0	REBA1.5	REBA2.0	REBA1.5	REBA2.0	REBA1.5	REBA2.0	REBA1.5
15	866	866	0	0	2841	2829	49	49
16	926.5	927	16	0	4197	4191	66	66
17	994.5	997	16	16	4082	4077	50	49
18	1420	1437.5	16	16	13668	10756	115	99
19	1861	1834.5	16	16	7783	7770	66	83
20	2103	2072	16	16	19634	14960	148	216
21	2345	2302.5	16	16	29137	24651	332	299
22	2703.5	2693	32	32	106636	39053	949	366
23	3457	3447	33	33	92259	80427	915	732

References

1. Allen, J.A., Minton, S.: Selecting the right heuristic algorithm: runtime performance predictors. In: Proceedings of 11th Biennial Conference of the Canadian Society for Computational Studies of Intelligence, pp. 41–53 (1996)
2. Beck, J.C.: Solution-guided multi-point constructive search for job shop scheduling. Journal of Artificial Intelligence Research 29, 49–77 (2007)
3. Borrett, J.E., Tsang, E.P.K.: A Context for Constraint Satisfaction Problem Formulation Selection. Constraints 6(4), 299–327 (2001)
4. Brélaz, D.: New methods to color the vertices of graphs. Communications of the ACM 22(4), 251–256 (1979)
5. Epstein, S.L., Freuder, E.C., Wallace, R., Morozov, A., Samuels, B.: The adaptive constraint engine. In: Van Hentenryck, P. (ed.) CP 2002. LNCS, vol. 2470, pp. 525–540. Springer, Heidelberg (2002)
6. Freuder, E.C.: A sufficient Condition for Backtrack-Free Search. Journal of ACM 29, 24–32 (1982)
7. Gagliolo, M., Schmidhuber, J.: Learning restart strategies. In: Veloso, M. (ed.) Proceedings, Twentieth International Joint Conference on Artificial Intelligence (IJCAI 2007), Hyderabad, India, January 6-12, pp. 792–797 (2007)
8. Gagliolo, M., Schmidhuber, J.: Impact of censored sampling on the performance of restart strategies. In: Benhamou, F. (ed.) CP 2006. LNCS, vol. 4204, pp. 167–181. Springer, Heidelberg (2006)
9. Gashnig, J.: A General Backtrack Algorithm That Eliminates Most Redundant Tests. In: Proceedings 5th International Joint Conference on Artificial Intelligence, vol. 457 (1977)
10. Gebruers, C., Hnich, B., Bridge, D., Freuder, E.: Using CBR to select solution strategies in constraint programming. In: Muñoz-Ávila, H., Ricci, F. (eds.) ICCBR 2005. LNCS, vol. 3620, pp. 222–236. Springer, Heidelberg (2005)
11. Gomes, C.P., Selman, B.: Algorithm portfolios. Artificial Intelligence 126(1-2), 43–62 (2001)
12. Gomes, C., Fernandez, C., Selman, B., Bessiere, C.: Statistical regimes across constrainedness regions. Constraints 10(4), 317–337 (2005)
13. Haralick, R.M., Elliott, G.L.: Increasing Tree Search Efficiency for Constraint Satisfaction Problems. Artificial Intelligence 14, 263–313 (1980)
14. Hoos, H., Tsang, E.P.K.: Local search for constraint satisfaction. In: Rossi, F., van Beek, P., Walsh, T. (eds.) Handbook of Constraint Programming, ch. 5, pp. 245–277. Elsevier, Amsterdam (2006)
15. Huberman, B., Lukose, R., Hogg, T.: An economics approach to hard computational problems. Science 265, 51–54 (1997)
16. Kern, M.: Parameter Adaptation in heuristic search - a population-based approach, PhD Thesis, Department of Computer Science, University of Essex, Colchester, UK (2005)
17. Kwan, A.: A framework for mapping constraint satisfaction problems to solution methods, PhD Thesis, Department of Computer Science, University of Essex, Colchester, UK (1997)
18. Mills, P., Tsang, E.P.K., Ford, J.: Applying an Extended Guided Local Search on the Quadratic Assignment Problem. In: Annals of Operations Research, vol. 118, pp. 121–135. Kluwer Academic Publishers, Dordrecht (2003)
19. Nudel, B.: Consistent-Labeling Problems and their Algorithms: Expected-Complexities and Theory-Based Heuristics. Artificial Intelligence 21, 135–178 (1983)
20. Prosser, P.: Hybrid Algorithms for the Constraint Satisfaction Problem. Computational Intelligence 9, 268–299 (1993)

21. Prosser, P.: Binary Constraint Satisfaction Problems: Some are Harder than Others. In: Proceedings 11th European Conference on Artificial Intelligence, pp. 95–99 (1994)
22. Puget, J.-F.: Applications of constraint programming. In: Montanari, U., Rossi, F. (eds.) Proceedings, Principles and Practice of Constraint Programming (CP 1995). LNCS, pp. 647–650. Springer, Heidelberg (1995)
23. Rossi, F., van Beek, P., Walsh, T. (eds.): Handbook of Constraint Programming. Elsevier, Amsterdam (2006)
24. Sabin, D., Freuder, E.C.: Contradicting Conventional Wisdom in Constraint Satisfaction. In: Proceedings 11th European Conference on Artificial Intelligence, pp. 125–129 (1994)
25. Smith, B., Grant, A.: Sparse Constraint Graphs and Exceptionally Hard Problems. In: Proceedings 14th International Joint Conference on Artificial Intelligence, pp. 646–651 (1995a)
26. Smith, B., Grant, A.: Where the *Exceptionally* Hard Problems are. In: Workshop on Studying and Solving Really Hard Problems. CP 1995, pp. 172–182 (1995b)
27. Smith, B.: Phase Transition and the Mushy Region in Constraint Satisfaction Problems. In: Proceedings 11th European Conference on Artificial Intelligence, pp. 100–104 (1994a)
28. Smith, B.: In search of Exceptionally Difficult Constraint Satisfaction Problems. In: Proceedings of the Workshop on Constraint Processing, 11th European Conference on Artificial Intelligence, pp. 79–86 (1994b)
29. Turner, J.S.: Almost all k-Colorable Graphs are Easy to Color. Journal of Algorithms 9, 63–82 (1988)
30. Tsang, E.P.K., Borrett, J.E., Kwan, A.C.M.: An Attempt to Map a Range of Constraint Satisfaction Algorithms and Heuristics. In: Proceedings AISB 1995, pp. 203–216 (1995)
31. Tsang, E.P.K.: Foundations of Constraint Satisfaction. Academic Press, London (1993)
32. Voudouris, C., Tsang, E.P.K.: Guided local search. In: Glover, F. (ed.) Handbook of metaheuristics, pp. 185–218. Kluwer, Dordrecht (2003)
33. Voudouris, C., Dorne, R., Lesaint, D., Liret, A.: iOpt: A Software Toolkit for Heuristic Search Methods. In: Walsh, T. (ed.) CP 2001. LNCS, vol. 2239, pp. 716–729. Springer, Heidelberg (2001)
34. Voudouris, C., Owusu, G., Dorne, R., Lesaint, D. (eds.): Service Chain Management: Technology Innovation for the Service Business. Springer, Heidelberg (2008)
35. Wolpert, D.H., Macready, W.G.: No Free Lunch Theorems for search, Technical Report SFI-TR-95-02-010, Santa Fe Institute (1995)
36. Wolpert, D.H., Macready, W.G.: No Free Lunch Theorems for Optimization. IEEE Transactions on Evolutionary Computation 1(1), 67–82 (1997)
37. Minton, S.: Automatically configuring constraint satisfaction programs, a case study. Constraints 1(1-2), 7–43 (1996)
38. Kautz, H., Horvitz, E., Ruan, Y., Gomes, C., Selman, B.: Dynamic restart policies. In: Proceedings, Eighteenth National Conference on Artificial Intelligence (AAAI 2002), Edmonton, Alberta, Canada, pp. 674–682 (2002)
39. Xu, L., Hutter, F., Hoos, H.H., Leyton-Brown, K.: SATzilla: portfolio-based algorithm selection for SAT. Journal of Artificial Intelligence Research 32, 565–606 (2008)