ECAI 2010
H. Coelho et al. (Eds.)
IOS Press, 2010
© 2010 The authors and IOS Press. All rights reserved.
doi:10.3233/978-1-60750-606-5-873

Learning When to Use Lazy Learning in Constraint Solving

Ian P. Gent[1] and **Chris Jefferson**[1] and **Lars Kotthoff**[1] and **Ian Miguel**[1] and **Neil C.A. Moore**[1] and **Peter Nightingale**[1] and **Karen Petrie**[2]

Abstract. Learning in the context of constraint solving is a technique by which previously unknown constraints are uncovered during search and used to speed up subsequent search. Recently, *lazy learning*, similar to a successful idea from satisfiability modulo theories solvers, has been shown to be an effective means of incorporating constraint learning into a solver. Although a powerful technique to reduce search in some circumstances, lazy learning introduces a substantial overhead, which can outweigh its benefits. Hence, it is desirable to know beforehand whether or not it is expected to be useful. We approach this problem using machine learning (ML). We show that, in the context of a large benchmark set, standard ML approaches can be used to learn a simple, cheap classifier which performs well in identifying instances on which lazy learning should or should not be used. Furthermore, we demonstrate significant performance improvements of a system using our classifier and the lazy learning and standard constraint solvers over a standard solver. Through rigorous cross-validation across the different problem classes in our benchmark set, we show the general applicability of our learned classifier.

1 Introduction

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

A *constraint satisfaction problem* (CSP [4]) is a set of decision variables, each with an associated domain of potential values, and a set of constraints. An assignment maps a variable to a value from its domain. Each constraint specifies allowed combinations of assignments of values to a subset of the variables. A *solution* to a CSP is an assignment to all the variables that satisfies all the constraints. Solutions are typically found for CSPs through systematic search of possible assignments to variables. During search, constraint *propagation* algorithms are used. These propagators make inferences, usually recorded as domain reductions, based on the domains of the variables constrained and the assignments that satisfy the constraints. If at any point these inferences result in any variable having an empty domain then search backtracks and a new branch is considered.

Propagation can dramatically reduce the space of assignments searched. Search can be further improved by the use of a lazy learning algorithm [11, 12, 13, 7], where previously unknown constraints are uncovered during search and used to speed up search subsequently. It is extremely efficient on some types of benchmark, but has a negative effect on others. Therefore, it is desirable to know beforehand whether or not lazy learning is expected to be useful.

We approach this problem using *machine learning*[3] to generate *decision trees*, which are a means of approximating discrete-valued target functions. These machine learning methods have been successfully applied to a broad range of tasks from learning to diagnose medical cases to learning to assess credit risk of loan applications [19]. In this paper, we show that decision trees can be built that successfully classify whether lazy learning should or should not be used on specific CP instances.

2 Background

We are addressing an instance of the Algorithm Selection Problem [22], which, given variable performance among a set of algorithms, is to choose the best candidate for a particular problem instance. Machine learning is an established method of addressing this problem [15, 16]. Particularly relevant to our work are the machine learning approaches that have been taken to configure, to select among, and to tune the parameters of solvers in the related fields of mathematical programming, propositional satisfiability (SAT), and constraints.

MULTI-TAC [18] configures a constraint solver for a particular instance distribution. It makes informed choices about aspects of the solver such as the search heuristic and the level of constraint propagation. The Adaptive Constraint Engine [5] learns (potentially novel) search heuristics from training instances. SATenstein [14] configures stochastic local search solvers for solving SAT problems.

An algorithm *portfolio* consists of a collection of algorithms, which can be selected and applied in parallel to an instance, or in some (possibly truncated) sequence. This approach has recently been used with great success in SATzilla [25] and CP Hydra [20]. In earlier work Borrett *et al* [3] employed a sequential portfolio of constraint solvers. Guerri and Milano [8] use a decision-tree based technique to select among a portfolio of constraint- and integer-programming based solution methods for the bid evaluation problem.

Rather than select among a number of algorithms, it is also possible to learn parameter settings for a particular algorithm. Hutter *et al* [10] apply this method to local search. Ansotegui *et al* [1] employ a genetic algorithm to tune the parameters of both local and systematic SAT solvers.

[1] University of St Andrews, Scotland, UK
[2] University of Dundee, Scotland, UK

[3] In this paper we inevitably use the word "learning" in two very different contexts with two very different meanings. These are machine learning and the lazy learning technique in constraint programming. We hope to minimise the possible confusion by using the phrases "machine learning" and "lazy learning" rather than abbreviating either to just "learning".

2.1 Lazy Learning in Constraints

Katsirelos *et al*'s [11, 12, 13] g-nogood learning (g-learning) is a notable CSP search algorithm. In short, whenever the solver reaches a dead-end state, a new constraint is added ruling out other branches that fail for a similar reason.

In order to achieve this, the first step is to analyse the earlier decisions and propagation that contributed to the current failure. We aim to find a set of assignments and disassignments that, if repeated, lead directly to a failure.

To analyse propagation, *explanations* are used:

Definition 1. *An* explanation for disassignment $x \not\leftarrow a$ *is a set of assignments and disassignments that are sufficient for a propagator to infer* $x \not\leftarrow a$. *Similarly an* explanation for assignment $y \leftarrow b$ *is a set of (dis-)assignments that are sufficient for a propagator to infer that* $y \leftarrow b$.

Example 1. Let a, b and c be three distinct values.

Suppose decision assignments $w \leftarrow a$ and $x \leftarrow a$ have been made. These assignments clearly also cause the remaining values of w and x to be ruled out; we can think of this disassignment being carried out by a built-in "at most one value" constraint. For example now $x \not\leftarrow b$ and the explanation for this disassignment is $\{x \leftarrow a\}$.

Now suppose the set of constraints includes both occurrence$([w, x, y, z], b) = 2$ and occurrence$([w, x, y, z], c) = 1$, meaning that variables w, x, y and z must have, respectively, exactly 2 occurrences of b and exactly 1 occurrence of c.

Since $w \leftarrow a$ and $x \leftarrow a$, the former constraint is forced to infer that $y \leftarrow b$ and $z \leftarrow b$. The explanation for both $y \leftarrow b$ and $z \leftarrow b$, for example, is $\{w \not\leftarrow b, x \not\leftarrow b\}$ because when w and x are both not assigned to b, we are forced to set the remainder of the variables to b.

Similarly, since $w \leftarrow a$, $x \leftarrow a$ and $y \leftarrow b$, the latter constraint is forced to infer that $z \leftarrow c$ in order to satisfy the constraint. The explanation for $z \leftarrow c$ is $\{w \not\leftarrow c, x \not\leftarrow c, y \not\leftarrow c\}$.

These explanations along with the decision assignments can be built into a *implication graph*:

Definition 2. *An* implication graph *for the current state of the variables is a directed acyclic graph where each node is a current (dis-)assignment and there is an edge from u to v iff u appears in the explanation for v.*

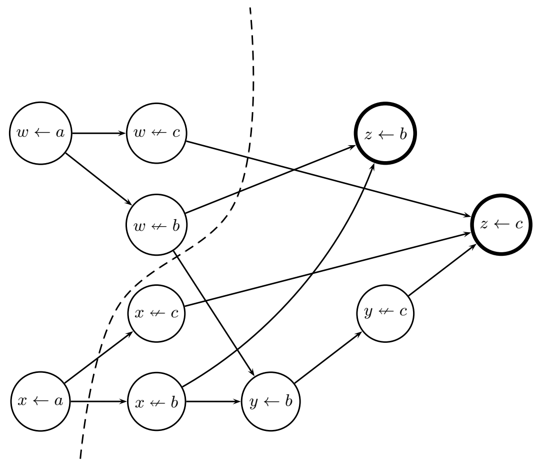

Figure 1. Implication graph for Example 1. Mutually inconsistent nodes shown with darkened nodes; cut from Example 2 with dashed line.

The explanations of Example 1 are displayed as an implication graph in Figure 1.

Since repeating the (dis-)assignments in an explanation will inevitably lead to the same propagation being repeated, repeating any cut of an implication graph for a failure will inevitably lead to the derivation of the failure again. Hence we build a constraint to avoid that failure by finding a cut $\{c_1, \ldots, c_k\}$ of the implication graph and then adding the constraint to avoid the failure $c = \neg(c_1 \wedge \ldots \wedge c_k)$. Now the solver backtracks and continues. For correctness and efficiency reasons we prefer certain cuts, but discussion of this issue is outwith the scope of this paper.

Example 2. The cut displayed as a dashed line in Figure 1 leads to the constraint $\neg(x \leftarrow a \wedge w \not\leftarrow c \wedge w \not\leftarrow b)$.

The power of g-learning comes from learned constraints proceeding to propagate and being combined by iterative application of the above process into more powerful constraints that can remove subtrees of the search tree, as opposed to just providing a shortcut to propagation, as in the above examples.

g-nogood learning is extremely effective on some types of benchmark, but has a negative effect on others. Firstly, there is an overhead associated with instrumenting constraint propagators to store explanations, which are needed to produce the new constraints. This problem is mitigated by *lazy learning* [7] which dramatically reduces the overhead of g-learning; however the new constraints must still be propagated and this slows the solver down, too.

Learning remains a high risk/high reward strategy – for many CSP instances, the overhead of learning is not justified by a decrease in nodes.

3 The Benchmark Instances

We evaluated the performance of the standard and the lazy learning solver on a set of 2028 benchmark instances from 46 different problem classes. The benchmark set has been chosen to include as many instances as possible whatever our expectation of which solver will work best. We were able to use only instances containing a subset of the following constraints: alldifferent, table, negative table, watched OR, lexicographic ordering, sum$\leq$, sum$<$, weightedsum$\leq$, weightedsum$<$, $x \leq y + c$, $\neq$, $x \leftarrow c$, $x \not\leftarrow c$, $\lfloor x/y \rfloor = z$, $x \bmod y = z$ and $x \times y = z$. See [2] for definitions of those for which we do not provide a citation. Our sources are Lecoutre's XCSP repository[4] and our own stock of CSP instances. For example, we include every extensional instance of the 2006 CSP solver competition and representatives from the random, industrial and academic spheres. Our set of instances is large, varied and inclusive.

In our experiments, we used the lazy learning variant of Minion, which we call Minion-lazy. The reference constraint solver used is Minion [6] version 0.9. A comparison of performance between Minion and Minion-lazy is given in Figure 2[5].

We imposed a time limit of 5000 seconds for each instance. For 11 instances, Minion-lazy ran out of memory (4GB) before it was able to solve the problem or reach the time limit. The total number of instances that neither solver could solve because of a time out or memory issues was 255. Both solvers took the same time on 4 instances that they could both solve.

The instances, the binaries to run them, and everything else required to reproduce our results is available at http://www.cs.st-andrews.ac.uk/~larsko/ecai2010/learning.tar.bz2.

[4] `http://tinyurl.com/y6hpphs`

[5] For this experiment we used binaries compiled with g++ version 4.4.1 and Boost version 1.38.0. The experiments were run on machines with 8 core Intel E5430 2.66GHz, 8GB RAM running CentOS with Linux kernel 2.6.18-164.6.1.el5 64Bit.

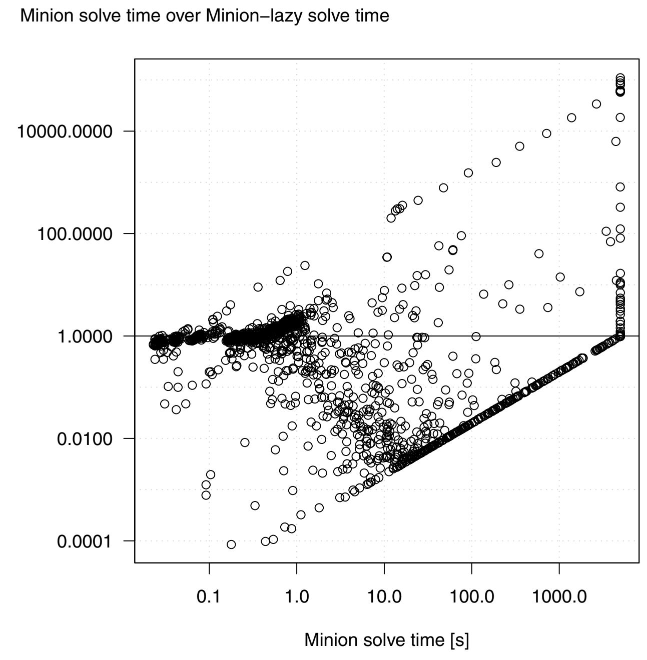

Figure 2. Scatterplot showing run time comparison for Minion-lazy vs Minion. Each point is a result for a single CSP. The x-axis is the solve time for Minion. The y-axis gives the speedup from using Minion-lazy instead of Minion. A ratio of 1 means they were the same, above 1 means Minion-lazy was faster and below 1 that Minion was faster.

4 Instance Attributes and their Measurement

We measured 85 attributes of the problem instances. They describe a wide range of features such as the number of constraints and variables used, a breakdown of the individual constraint and variable types and a number of attributes based on the primal graph. The primal graph $g = \langle V, E\rangle$ has a vertex for every CSP variable, and two vertices are connected by an edge iff the two variables are in the scope of a constraint together.

For space reasons, we cannot provide a description of all the attributes we measured. We focus instead on the attributes that we have identified as reflecting the structure of a constraint problem; in particular they do not depend on particular solver features or are specific to particular problem classes.

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

Clustering coefficient For a vertex v, the set of neighbours of v is $n(v)$. The edge density among the vertices $n(v)$ is calculated. The clustering coefficient is the mean average of this local edge density for all v [23]. It is intended to be a measure of the local cliqueness of the graph. This attribute has been used with machine learning for a model selection problem in constraint programming [8].

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

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

Width of ordering Each of our benchmark instances has an associated variable ordering. The width of a vertex v in an ordered graph is its number of *parents* (i.e. neighbours that precede v in the ordering). The width of the ordering is the maximum width over all vertices [4] (ch. 4). The width of the ordering and the width normalised by the number of vertices were used.

Width of graph The width of a graph is the minimum width over all possible orderings. This can be calculated in polynomial time [4], and is related to some tractability results. The width of the graph and the width normalised by the number of vertices were used.

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

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

Normalised SAC literals The number of literals pruned by singleton consistency preprocessing, as a proportion of all literals.

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

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

Literal tightness To measure the tightness of a literal (a variable-value pair) w.r.t. a particular constraint, we sample 100 random tuples containing the literal and test if the tuples satisfy the constraint. The tightness of a literal is the mean of its tightness in all constraints on that literal. The mean literal tightness – the mean average of the tightness for each literal – and the standard deviation of the literal tightness divided by the mean literal tightness are used (a.k.a. the coefficient of deviation).

Proportion of symmetric variables In many CSPs, the variables form equivalence classes where the number and type of constraints a variable is in are the same. For example in the CSP $x_1 \times x_2 = x_3, x_4 \times x_5 = x_6, x_1, x_2, x_4, x_5$ are all indistinguishable, as are x_3 and x_6. The first stage of the algorithm used by Nauty [17] detects this property. Given a partition of n variables generated by this algorithm, we transform this into a number between 0 and 1 by taking the proportion of all pairs of variables which are in the same part of the partition.

In creating this set of attributes, we intended to cover a wide range of possible factors in the success of lazy learning. Wherever possible, we normalised attributes that would be specific to problem instances of a particular size, such as the number of variables. This is based on the intuition that similar instances of different sizes are likely to behave similarly with lazy learning.

Of the 2028 instances from the benchmark set, 93 instances could not be analysed because of insufficient memory or unsupported constructs in the input.

5 Constructing an Instance Classifier using Machine Learning

In this section, we construct a number of classifiers using standard machine learning techniques. First we describe the ML methodology we used.

5.1 Methodology

The experimental results were preprocessed to perform two kinds of filtering. We do not want instances where the relative difference between the two solvers is very small to affect the classifier. Also we want instances where the penalty for making the wrong decision is high to be more important than ones where the penalty is low.

First, we calculated the misclassification penalty as the absolute difference in solve time between the two solvers for each instance. Instances where both solvers timed out were not considered. For instances where one of the solvers timed out, we used the timeout (5000s) as the time for that solver, and calculated the misclassification penalty as before. The resulting value is an underestimate of the true misclassification penalty.

Second, we determined which solver to use for each instance. If the difference between both solvers was less than 20% of the time that Minion took, we set the value to "don't know". This was simply to account for differences in run time that are caused by external effects. For the instances where Minion-lazy ran out of memory, we set the value to "use Minion". Based on this, we were able to make a decision for 1012 instances.

Each instance was then weighted by the misclassification penalty. We did this to bias the machine learning algorithms towards the instances where we can gain or lose a lot – if the penalty for choosing the wrong solver is low, we do not care as much if we make the wrong decision. To achieve this cost-sensitive classification, it is standard practice in machine learning to duplicate instances [24]. We duplicated each instance $\lceil \log_2(\texttt{misclassification penalty}) \rceil$ times. This means that instances with a misclassification penalty of less than 2 seconds appear once and ones with a penalty of 5000 seconds (the maximum) appear 13 times. We chose this particular function because we did not want instances with a very high penalty to have too much effect and we did want each instance to appear at least once in the data set. The instances where Minion-lazy ran out of memory were not duplicated.

We used this data as input for the WEKA [9] machine learning suite to learn a classifier that, given an instance, tells us which solver to use. For instances where we did not have values for all attributes or some of them were unclear, for example when we were not able to make a decision between choosing Minion and Minion-lazy, we used a question mark.

The WEKA ML algorithm we used was `J48`, which creates decision tree classifiers. `J48` implements the well-established C4.5 algorithm [21]. The default values for the parameters gave good performance already and were therefore not tweaked. In all cases we trained the classifiers to predict the binary decision of whether to use Minion or Minion-lazy.

We evaluated the learned classifiers on the whole set of instances, making the decision which solver to use for each one. If we were unable to make a decision because of missing attributes, we defaulted to Minion. The results of this evaluation are different from the ones we get in WEKA because we use the data set that contains each instance only once – we train and test on the data set that we have biased towards instances where we gain or lose a lot by duplicating them, and we evaluate on the unbiased original set. The results are summarised in Table 1.

5.2 Selecting an Appropriate Attribute Set

Finding a small and appropriate set of attributes is crucial to the efficiency of classifying an instance, since all attributes must be computed. In this section, we describe four classifiers built with progressively smaller sets of attributes. We demonstrate that the final classifier, which selected from 3 attributes, is almost as accurate as the first, which selected from 85 attributes.

Classifier 1 — Initial Classifier. We built a decision tree using the whole data set with 85 attributes as described above. The tree had 61 nodes and we achieved 99.7% correctly classified instances with a precision of 99.7% and a recall of 99.7%. For comparison, always using standard Minion gives 86.3% correctly classified instances, a precision of 74.4% and a recall of 86.3%.

The performance of the learned classifier on the set of all benchmark instances is summarised in Table 1, labelled as *classifier 1*. This demonstrates the feasibility of our approach of using machine learning to make this decision.

Classifier 2 — Reduced Number of Attributes. Based on the encouraging results we achieved with classifier 1, we removed all but the 17 attributes described in Section 4 and reran the decision tree building algorithm.

Computing a large number of attributes for each instance is expensive and might outweigh the benefit of having a classifier and being able to select the faster solver. We also want to eliminate the influence of attributes which are specific to Minion, a problem class, or a particular problem size.

The classifier learned from this data set showed a similar performance to the previous one. The decision tree had 57 nodes and 99.6% of the instances were classified correctly. Precision and recall were 99.6% again as well. Table 1 however shows that the overall performance is lower than the one of the previous classifier, although the difference is small (classifier 2). We concluded that we could reduce the number of attributes used in making the decision without a significant decrease in quality.

Classifier 3 — Cheaply Computable Attributes. We are not only interested in decision trees which use a small number of attributes, the attributes must be computable in a short time as well. For classifier 3, we eliminated the two attributes multiple shared variables and proportion of symmetric variables, which are particularly expensive to compute, from our data set. The learned decision tree consisted of 65 nodes and again gave 99.6% correctly classified instances and 99.6% precision and recall. Table 1 shows that the overall performance decreases very slightly compared to classifier 2.

Classifier 4 — Most Influential Attributes. None of the previous classifiers used all the attributes in the data set. We decided to further reduce the number of attributes used in the decision tree by using the WEKA `AttributeSelectedClassifier` metalearning algorithm.

Before letting the `J48` algorithm build a decision tree, we ran the `CfsSubsetEval` attribute selector with exhaustive search. Each attribute was assessed with respect to its predictive ability and the degree of redundancy within the subset of attributes for all subsets of the previous set of 15 attributes [24]. Only three attributes where selected; normalised width of ordering (NWO), normalised mean constraints per variable (NMCV) and mean tightness (MT). Based on the application of this attribute selection algorithm, we are confident that the final set of attributes does not contain irrelevant or redundant ones.

The decision tree built with these three attributes had 79 nodes, 99.6% correctly classified instances and precision and recall of 99.6%. As Table 1 shows, the performance is similar to classifier 3.

5.3 Eliminating Overfitting

The decision tree of classifier 4 appears to be overfitted. Specifically, for most paths from the root to a leaf node, it switches several times on a single parameter. For example, when $0.101 < \text{NWO} \leq 0.124$ and $\text{NMCV} > 0.003$, the decision tree then branches on mean tightness (MT). It matches the intervals $[0, 6.79\%](6.79\%, 6.81\%](6.81\%, 23.59\%](23.59\%, 100\%)$ of MT to decisions *yes*, *no*, *yes*, *no* respectively. The interval

	instances solved	total time [s]	WEKA accuracy [%]	misclassified	misclassified > 20% difference	misclassified penalty [s]	compute features [s]
oracle	1,773	194,372	-	0	0	0	-
anti-oracle	1,485	1,805,732	-	1,769	1,046	1,611,360	-
Minion	1,736	360,949	86.3	391	292	166,577	0
Minion-lazy	1,522	1,639,155	13.7	1,378	754	1,444,783	0
Classifier 1	1,769	201,726	99.7	267	30	7,354	126,917
Classifier 2	1,767	207,327	99.6	316	50	12,955	124,918
Classifier 3	1,766	211,844	99.6	314	48	17,472	122,957
Classifier 4	1,767	214,176	99.6	360	75	19,804	11,285
Classifier 5	1,765	231,531	96.8	399	97	37,159	11,285

Table 1. Summary of classifier performance. WEKA accuracy denotes the average percentage of correctly classified instances during cross-validation. The oracle classifier always makes the right decision, the anti-oracle the wrong decision, Minion always picks the standard solver and Minion-lazy always the constraint learning solver. The time to compute the features is the total time for all the 1,935 instances that could be analysed. The total time is the time taken to process all 1,773 instances which either solver can solve, including 5,000s for each instance which a given classifier fails to solve. We exclude the 255 instances where both solvers time out, which would add 1,275,000s to each classifier. An instance is misclassified if at least one solver solves it within the time limit and there is a solver other than the chosen one which took less time. The misclassification penalty is the number of seconds the specific classifier took longer than the oracle would have taken. All times rounded to the nearest second.

$(6.79\%, 6.81\%]$ is a clear case of overfitting – it is very narrow and contains only one instance from the original data set. The instance occurs 4 times in the training set because of its misclassification penalty.

Classifier 5. To eliminate overfitting, we tune the parameters of the ML algorithm to prune the tree more aggressively. The result is a much smaller decision tree without the apparent overfitting.

We adjust the parameters of the `J48` algorithm to perform more pruning of the learned tree. Specifically, we reduced the confidence threshold for pruning from 25% to 1% and increased the minimum number of instances permissible at a leaf from 2 to 50. The decision tree generated with these parameters has only 13 nodes and is depicted in Figure 3. The classifier evaluated 96.8% of instances correctly and had a precision of 96.8% and a recall of 96.8%. As Table 1 shows, the performance clearly suffers. However, as we discuss in detail in Section 6, the losses in practical performance are small compared to the win over using standard Minion, and with a classifier that is much less likely to be overfitted.

5.4 Testing the Generality of Classifier 5

We provide evidence for the general applicability of classifier 5. For unknown problem instances, we have to consider two cases. The new instance could belong to a problem class that is contained in our data set. In this case, we are confident that our classifier will give a good result based on its high accuracy on the existing benchmark set, which would contain similar instances. In the more interesting case, a new instance belongs to a problem class of which no instances are contained in our set of benchmarks.

To evaluate the performance of our classifier on unknown problem classes, we take the well-established principle of leave-one-out cross-validation and apply it to the set of problem classes. For n-fold cross-validation, the original data set is split into n parts of roughly equal size. Each of the n partitions is in turn used for testing. The remaining $n - 1$ partitions are used for training. In the end, every instance will have been used for both training and testing in different runs [24]. Leave-one-out cross-validation is n-fold cross-validation where n is the size of the data set.

We create new data sets, each with a particular problem class removed. The idea behind it is that if the `J48` algorithm, using the same methodology as described above, produces the decision tree shown in Figure 3 for subsets of the problem classes, then it is likely to be problem class-independent. Unfortunately, not all of the problem classes have both instances where lazy learning is faster and slower in our benchmark set. Leaving one of these problem classes out may not have any effect on the learned decision tree. On the other hand, instances from an unknown problem class may have the same characteristic. For 24 of 46 problem classes, the instances can be split into ones where lazy learning is faster and ones where it is slower.

Out of the 46 generated sets, we got exactly the same tree as shown in Figure 3 for 24. 18 of those sets were created by removing a problem class that has no instances where Minion-lazy is faster. For all of the decision trees, the attribute to switch on at the root node was the same and for all but one of them the value to switch on was the same as well. For 15 out of the 22 classifiers that were different, the only modification was the addition or deletion of a single subtree. Typical trees which were generated several times are shown in Figures 4 and 5. They are still very similar to the classifier in Figure 3; the differences are highlighted with dashed lined. For all but two of the generated trees, the proportion of correctly classified instances was higher than 95% and higher than 90% for all of them.

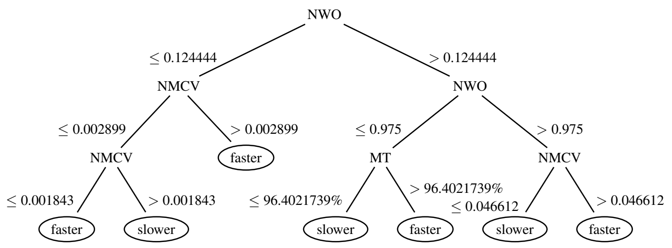

Figure 3. Final decision tree (classifier 5) to predict if, for a given instance, the solver with lazy learning will be faster or slower. NWO stands for normalised width of ordering, NMCV for normalised mean constraints per variable and MT for mean tightness.

6 Performance of the General Decision Tree

The performance of the final decision tree in Figure 3 improves significantly over simply running standard Minion on all of the instances, as shown in Table 1. In particular, it achieves 29 more instances solved in 129,418 seconds less, compared to a maximum possible gain of 37 more instances in 166,577 seconds less for the perfect oracle classifier.

Calculating the attributes the classifier requires takes approximately 6 seconds per instance and we gain more than 70 seconds on average by using the classifier. We are therefore left with a net

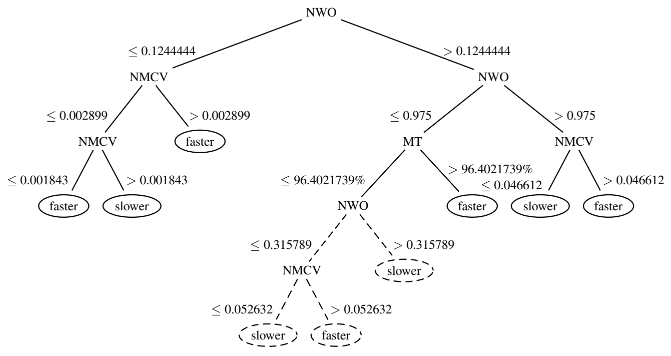

Figure 4. Typical decision tree with an additional subtree that was generated during cross-validation with data sets with one problem class left out. The added subtree is highlighted with dashed lines.

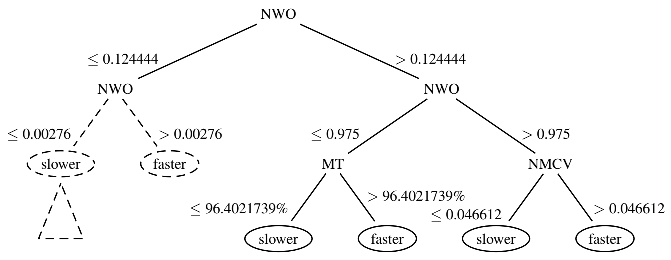

Figure 5. Typical decision tree with a subtree pruned that was generated during cross-validation with data sets with one problem class left out. The missing subtree is highlighted with dashed lines. Note that also the attribute that is switched on above the pruned subtree is different.

win over the whole benchmark set of approximately 39 hours for a total run time of approximately 422 hours. This is almost 85% of the best possible improvement; the oracle classifier gives a win of about 46 hours, even assuming that it required no attributes to make its decision and ran in zero time.

This shows that through careful tuning of the parameters used in the decision tree, using our classifier provides a significant win even when taking the overhead of calculating the attributes into account. The decision algorithm itself is very simple and the effective cost of making the decision once the attribute values are known is tiny.

7 Conclusions

In this paper, we have applied machine learning techniques to constraint solving to decide whether or not to use lazy learning, a powerful but costly technique. To facilitate this, we have used a large set of benchmarks from many different problem classes. We have demonstrated the success of our approach by learning classifiers with an extremely high accuracy that, when applied to our set of benchmarks, are able to solve many more instances in significantly less time than the standard solver alone. In particular, they are very close to the perfect oracle classifier.

We identified two problems with our initial approach: that attributes needed by classifiers would be expensive to compute, and the resulting classifier was overfitted to our benchmark set. We addressed these problems by reducing the number of attributes used and the size of the decision tree to provide decision algorithms that can be computed cheaply. The result is a tree with only six internal nodes, and thus unlikely to be overfitted, which still gets about two-thirds of the improvement that would be obtained with a zero-cost oracle. We have furthermore evaluated and provided substantial evidence for the general applicability of our learned classifier by cross-validating it over different problem classes. We suggest therefore that the decision tree shown in Figure 3 is very likely to make good decisions for problem instances that are not included in our set of benchmarks.

ACKNOWLEDGEMENTS

The authors thank Alan Frisch and James Cussens for helpful comments on an earlier version of this paper and general discussions about machine learning. We also thank the anonymous reviewers for their feedback. Chris Jefferson and Peter Nightingale are supported by EPSRC grant EP/H004092/1. Neil Moore is supported by EPSRC grant EP/E030394/1. Karen Petrie is supported by a Royal Society Dorothy Hodgkin Research Fellowship. Lars Kotthoff is supported by a SICSA studentship.

REFERENCES

[1] C. Ansótegui, M. Sellmann, and K. Tierney, 'A gender-based genetic algorithm for the automatic configuration of algorithms', in *CP*, pp. 142–157, (2009).

[2] N. Beldiceanu, M. Carlsson, and J.-X. Rampon, 'Global constraint catalog', Technical Report 08, Swedish Inst. of Comp. Sci., (2005).

[3] J.E. Borrett, E.P.K. Tsang, and N.R. Walsh, 'Adaptive constraint satisfaction: The quickest first principle', in *ECAI*, pp. 160–164, (1996).

[4] R. Dechter, *Constraint Processing*, Elsevier Science, 2003.

[5] S.L. Epstein, E.C. Freuder, R.J. Wallace, A. Morozov, and B. Samuels, 'The adaptive constraint engine', in *CP*, pp. 525–542, (2002).

[6] I.P. Gent, C. Jefferson, and I. Miguel, 'Minion: A fast scalable constraint solver.', in *ECAI*, pp. 98–102, (2006).

[7] I.P. Gent, I. Miguel, and N.C.A. Moore, 'Lazy explanations for constraint propagator', in *PADL 2010*, pp. 217–233, (2010).

[8] A. Guerri and M. Milano, 'Learning techniques for automatic algorithm portfolio selection', in *Proc. ECAI 2004*, pp. 475–479, (2004).

[9] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I.H. Witten, 'The WEKA data mining software: An update', *SIGKDD Explorations*, **11**(1), (2009).

[10] F. Hutter, Y. Hamadi, H.H. Hoos, and K. Leyton-Brown, 'Performance prediction and automated tuning of randomized and parametric algorithms', in *CP*, pp. 213–228, (2006).

[11] G. Katsirelos, *Nogood Processing in CSPs*, Ph.D. dissertation, University of Toronto, Jan 2009.

[12] G. Katsirelos and F. Bacchus, 'Unrestricted nogood recording in CSP search', in *CP*, pp. 873–877, (2003).

[13] G. Katsirelos and F. Bacchus, 'Generalized nogoods in CSPs', in *AAAI*, pp. 390–396, (2005).

[14] A.R. KhudaBukhsh, L. Xu, H.H. Hoos, and K. Leyton-Brown, 'SATenstein: Automatically building local search SAT solvers from components', in *IJCAI*, pp. 517–524, (2009).

[15] M.G. Lagoudakis and M.L. Littman, 'Reinforcement learning for algorithm selection', in *AAAI/IAAI*, p. 1081, (2000).

[16] K. Leyton-Brown, E. Nudelman, G. Andrew, J. McFadden, and Y. Shoham, 'A portfolio approach to algorithm selection', in *IJCAI*, pp. 1542–1542. Morgan Kaufmann, (2003).

[17] B. McKay, 'Practical graph isomorphism', in *Num. math. and comp., 10th Manitoba Conf., Congr. Numerantium 30*, pp. 45–87, (1981).

[18] S. Minton, 'Automatically configuring constraint satisfaction programs: A case study', *Constraints*, **1**(1/2), 7–43, (1996).

[19] T.M. Mitchell, *Machine Learning*, McGraw-Hill, 1997.

[20] E. O'Mahony, E. Hebrard, A. Holland, C. Nugent, and B. O'Sullivan, 'Using case-based reasoning in an algorithm portfolio for constraint solving', in *19th Irish Conference on AI*, (2008).

[21] J.R. Quinlan, *C4.5: Programs for Machine Learning*, Morgan Kaufmann, 1993.

[22] J.R. Rice, 'The algorithm selection problem', *Advances in Computers*, **15**, 65–118, (1976).

[23] D.J. Watts and S.H. Strogatz, 'Collective dynamics of 'small-world' networks', *Nature*, **393**, 440–442, (1998).

[24] I.H. Witten and E. Frank, *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*, Morgan Kaufmann, 2nd edn., 2005.

[25] L. Xu, F. Hutter, H.H. Hoos, and K. Leyton-Brown, 'SATzilla: Portfolio-based algorithm selection for SAT', *J. Artif. Intell. Res. (JAIR)*, **32**, 565–606, (2008).