

# Learning When to Use Lazy Learning in Constraint Solving

CIRCA preprint 2010/02

Ian P. Gent and Lars Kotthoff and Ian Miguel and Neil C.A. Moore and Peter Nightingale  
University of St Andrews

Karen Petrie  
University of Dundee

## 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 search subsequently. 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 in a constraint solver. Although a powerful technique to reduce search in some circumstances, lazy learning does introduce a substantial overhead which can outweigh its benefits. Therefore, 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 classifiers which perform well in identifying instances on which constraint learning should or should not be used. Furthermore, we were able to identify mean constraint tightness as an interesting attribute that plays a key role in identifying instances where lazy learning performs well.

## Introduction

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

A *constraint satisfaction problem* (CSP (Dechter 2003)) 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 found for CSPs through systematic search of possible assignments to variables. During search constraint *propagation* algorithms are used. These propagators make inferences, 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.

Copyright © 2010, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

Propagation can dramatically reduce the space of assignments searched. Search can be further improved by the use of a constraint learning algorithm, where previously unknown constraints are uncovered during search and used to speed up search subsequently. *Lazy learning* (Katsirelos 2009; Katsirelos and Bacchus 2003; 2005; Gent, Miguel, and Moore 2010) is an up-to-date CSP search algorithm. Lazy learning 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*<sup>1</sup> to generate *decision algorithms* in the form of rules or trees. These are means for 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 (Mitchell 1997).

In this paper, we show that decision algorithms can be built which successfully classify whether lazy learning should or should not be used on specific CP instances. In particular we identified mean constraint tightness as an interesting attribute of a CSP that plays a key role in identifying instances where lazy learning performs well.

In the next section, we go on to explain more about the background of both the machine learning and CSP techniques used in this paper. In the subsequent section we discuss which of the attributes of a CSP we considered in our experiments. After that, we discuss the experiments, how we used the results to generate classifiers using machine learning, and how we evaluated the findings on a different set of benchmarks. The paper closes by discussing the importance of the results we have obtained.

## Background

Essentially we are addressing an instance of the Algorithm Selection Problem (Rice 1976), which, given variable performance among a selection of algorithms, is to

<sup>1</sup>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”.

decide which is the best candidate for a particular problem instance. Machine Learning is an established method of addressing this problem (Lagoudakis and Littman 2000; Leyton-Brown et al. 2003). 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 closely-related fields of mathematical programming, propositional satisfiability (SAT), and constraints.

Minton’s MULTI-TAC system (Minton 1996) synthesizes a constraint solver for a particular instance distribution. It is able to make informed choices about aspects of the solver such as the search heuristic and the level of constraint propagation. The Adaptive Constraint Engine (Epstein et al. 2002) learns (potentially novel) search order heuristics for training instances. SATenstein (KhudaBukhsh et al. 2009) 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 (Xu et al. 2008). In earlier work Borrett *et al* (Borrett, Tsang, and Walsh 1996) employed a sequential portfolio of constraint solvers. Guerri and Milano (Guerri and Milano 2004) 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* (Hutter et al. 2006) apply this method to local search. Ansotegui *et al* (Ansotegui, Sellmann, and Tierney 2009) employ a genetic algorithms to tune the parameters of both local and systematic SAT solvers.

## Lazy Learning in Constraints

Katsirelos *et al*’s (Katsirelos 2009; Katsirelos and Bacchus 2003; 2005) 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  $x \leftarrow a$  is a set of assignments and disassignments that are sufficient for a propagator to infer  $x \leftarrow a$ . 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  $x \leftarrow b$  and the explanation for this disassignment is  $\{x \leftarrow a\}$ .

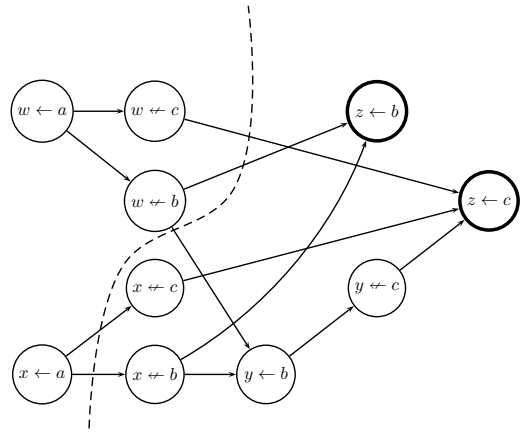


Figure 1: Implication graph for Example 1. Mutually inconsistent nodes shown with darkened nodes; cut from Example 2 with dashed line.

Now suppose the set of constraints includes both  $\text{occurrence}([w, x, y, z], b) = 2$  and  $\text{occurrence}([w, x, y, z], c) = 1$ , 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  $\{w \leftarrow b, x \leftarrow b\}$  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  $\{w \leftarrow c, x \leftarrow c, y \leftarrow c\}$ .

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  $v$ .

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  $\{c_1, \dots, c_k\}$  of the implication graph, and then the constraint to avoid the failure is  $c = \neg(c_1 \wedge \dots \wedge c_k)$ . Now  $c$  is added, 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  $\neg(x \leftarrow a \wedge w \leftarrow c \wedge w \leftarrow b)$ .

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 op-

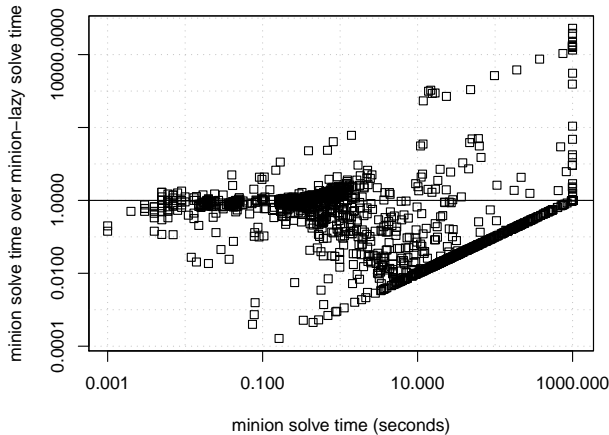


Figure 2: Scatterplot showing runtime comparison for Minion-lazy vs Minion. Each point is a result for a single CSP. The  $x$ -axis is the solve time for Minion (i.e., excluding set up time which is identical for both). The  $y$ -axis gives the speedup from using Minion-lazy instead of Minion. A ratio of 1 means they were the same, above 1 means Minion-lazy was faster and below 1 that Minion was faster.

posed 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* (Gent, Miguel, and Moore 2010) which dramatically reduces the overhead of g-learning; however the new constraints must still be propagated and this slows the solver down, too.

In our experiments we will use the lazy learning variant of Minion, which we call Minion-lazy. The reference constraint solver used is Minion (Gent, Jefferson, and Miguel 2006). A comparison of performance between Minion and Minion-lazy is given in Figure 2<sup>2</sup>. This graph illustrates that learning remains a high risk/high reward strategy: for many CSP instances the overhead of learning is not justified by a decrease in nodes.

## Instance Attributes and their Measurement

We used 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.

<sup>2</sup>For this experiment we used binaries compiled with g++ version 4.4.1 and Boost version 1.38.0. The experiments were run 8 in parallel on machines with 8 core Intel E5430 2.66GHz, 8GB RAM running CentOS with kernel 2.6.18-164.6.1.el5 64Bit. Links to source code removed to preserve anonymity.

**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  $v$  (Watts and Strogatz 1998). 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 (Guerri and Milano 2004).

**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 (Dechter 2003) (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 (Dechter 2003), and is related to some tractability results. The width of the graph and the width normalised by the number of vertices were used.

As well as attributes based on the primal graph, we employed a number of other attributes, ranging from properties of the constraint hypergraph to symmetry.

**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.

**Mean literal tightness:** To measure the tightness of a literal 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 is the mean average of the tightness for each literal.

**Literal tightness coefficient of variation:** This is the standard deviation of the literal tightness divided by the mean literal tightness.

**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 (McKay 1981) 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. However, 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.

We also excluded attributes about the proportion of different constraint types, preferring to summarise the constraints with information about tightness, symmetry and overlap between constraints. This is to avoid the classifier identifying particular problem classes by the proportions of constraints, as the intention is to classify instances.

## The Benchmark Instances

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,  $\text{sum} \leq$ ,  $\text{sum} <$ ,  $\text{weightedsum} \leq$ ,  $\text{weightedsum} <$ ,  $x \leq y + c$ ,  $\neq$ ,  $x \leftarrow c$ ,  $x \nleftarrow c$ ,  $\lfloor x/y \rfloor = z$ ,  $x \bmod y = z$  and  $x \times y = z$ . See (Beldiceanu, Carlsson, and Rampon 2005) for definitions of those that we do not provide a citation for. Our sources are Lecoutre’s XCSP repository (Lecoutre 2010) 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 unbiased. Instances will be published online but link removed to preserve anonymity.

## Constructing an Instance Analyser using Machine Learning

We applied several stages of filtering to the experimental results<sup>3</sup>. First, we removed the instances which took less than two seconds to solve to account for variations in run time due to the reading of the input, initial memory allocation, and similar things.

Second, we discarded the instances where it was not clear whether Minion-lazy or Minion was faster. We required a difference in run time of at least 50% between the two solvers to account for a certain margin of experimental error

<sup>3</sup>For this experiment we used binaries compiled with g++ version 4.4.1 and Boost version 1.38.0. The experiments were run 4 in parallel on machines with 4 core Intel Q6600 2.4GHz, 4GB RAM running CentOS with kernel 2.6.18-128.7.1.el5 64Bit.

and external influences like the effect of other processes on the CPU cache.

Out of a total of 2024 instances which were run, 519 were available for evaluation after filtering.

We used the WEKA (Hall et al. 2009) machine learning toolbox to determine the problem attributes which affect the performance of Minion-lazy.

Minion-lazy performed better than Minion on 42 of the 519 instances we used for the analysis. To avoid biasing WEKA towards the instances where Minion-lazy was slower and to have a wider range of training sets, we selected 100 different training sets containing the 42 “positive” instances and the same number of “negative” instances chosen at random. We used the full set of instances for testing the classifiers.

WEKA provides a plethora of classifiers. We chose five which create decision algorithms of different complexity and with a high precision and recall.

**OneR** As one of the most basic classifiers, OneR chooses one attribute to switch on. It classifies the instance based on the range of the value of the attribute.

**ConjunctiveRule** This classifier uses more than one attribute for classification. The instance must satisfy a conjunction of attributes and ranges of values.

**ADTree** This classifier builds an alternating decision tree.

**REPTree** Builds a regression tree using information gain criteria.

**J48** The J48 classifier builds a decision tree using the C4.5 algorithm where the levels are different attributes and the nodes are different values of those attributes.

For all classifiers, the parameters were left at their default values. We used a “black box” approach to WEKA to see whether someone with no previous experience in machine learning could use it for decision making in constraint programming.

For each classifier/training set combination, we extracted the attribute the classifier considered to be the most important one and precision and recall of the decision algorithm on the test set.

For all of the experiments, the precision on the test set was higher than 90%. The recall was similarly high, averaging between 85% and 90%. There was no significant difference in terms of precision and recall between the different classifiers.

No attribute was always the most important one across all classifiers and training sets. There was a variety of different decision rules and trees that the classifiers came up with. The attributes that the algorithms considered to be the most important ones most often are as follows (from most to least often):

- mean tightness;
- normalised width of ordering;
- normalised standard deviation of node degree;
- normalised mean degree; and
- normalised mean constraints per variable.

Figure 3 shows a typical decision tree that the J48 classifier computed for one of the training sets. It is typical in the sense that most decision trees used mean tightness at the root node and had approximately the same number of levels.

We observed that the precision and recall for decision algorithms that use just one attribute were typically as good as for more complex decision algorithms with more than one attribute. Also, simple decision algorithms with one attribute are preferable because each attribute must be computed before a decision is made. These considerations caused us to focus on one attribute, the mean tightness, from here on.

Based on the results, we investigated the value of the mean tightness where the transition from “Minion-lazy faster” to “Minion faster” occurs. Most of the values chosen by the classifiers were between 6% and 7% – this suggests that for problems with a mean tightness of less than 6%, Minion-lazy should perform better than Minion.

### Evaluation of the importance of mean tightness

As stated above, mean tightness was the most important attribute in many decision trees and rules produced by the machine learning algorithms. It was the most important attribute most often, across different classifiers and training sets. This leads us to the hypothesis that mean tightness is an important attribute in general, not simply for our benchmark set.

To test this hypothesis we generated a new set of random problem instances. This allowed us to vary mean tightness while fixing other parameters. We used Model RB described in (Lecoutre 2009). The model has parameters  $k, n, \alpha, r, p$  where  $k$  is the arity,  $n$  is the number of variables,  $\alpha$  determines the domain size (via the formula  $d = n^\alpha$ ),  $r$  determines the number of constraints (via the formula  $e = r n \ln n$ ) and  $p$  is the tightness of all constraints. In all cases, we use  $\alpha = 1$  so the domain size equals the number of variables. We generated instances with arity 3 and 12 and 13 variables, and with arity 4 and 10 variables. The chosen values for the mean tightness  $p$  ranged from 0.002 to 0.01 in 0.001 increments, and from 0.01 to 0.2 in 0.01 increments.  $r$  was calculated to be at the satisfiability phase transition with the formula  $r = -\alpha / \ln(1 - p)$ . For each combination of  $k, n$  and  $p$ , 10 random instances were generated for a total of 870 instances. We ran the random instances with Minion and with Minion-lazy, with a timeout of 1000 seconds<sup>4</sup>.

As  $p$  is decreased, the instances become increasingly difficult and contain a very large number of constraints. As the tightness of each constraint is decreased, the number of constraints must increase to remain at the phase transition.

Figure 4 shows our results for this experiment. Where mean tightness is low, Minion-lazy tends to perform comparatively well on the instances that could be solved by either solver. In this region, a large proportion of the instances caused both solvers to time out. Minion-lazy is most suc-

<sup>4</sup>For this experiment we used the same machines and binaries as for the earlier experiment described in Footnote 2.

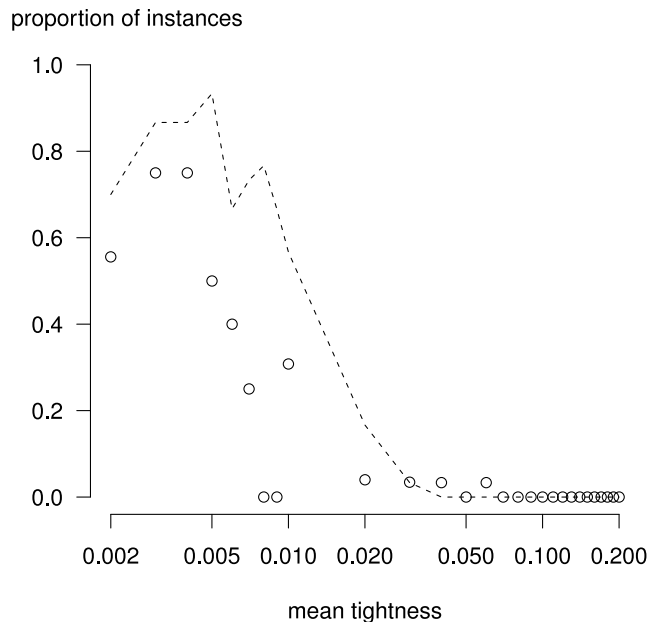


Figure 4: A plot of our results on random instances. The x axis is the mean tightness on a log scale. The circles are the proportion of instances where lazy learning is faster, from those where at least one solver finished. The dashed line shows the proportion of instances where both solvers timed out (beyond 1000 seconds).

cessful where the instances are difficult for both solvers, which reflects well on lazy learning.

Over the whole set of 870 instances, lazy learning does not perform well. It is not a surprise that lazy learning is comparatively poor on random instances, as this has been observed before on the g-learning (Katsirelos 2009) as well as conflict-clause driven SAT solvers (Various 2010).

These results support our hypothesis that mean tightness is an important factor in the success of lazy learning, and that a low value of this attribute tends to be good for lazy learning. However, the cut-off value is not the 6% that the results from the previous section would suggest, but a much lower value. It appears that it varies for different problem classes, and therefore that there are other important factors.

### Conclusion and Future Work

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 found strong evidence which suggests that the mean tightness of a constraint problem is one of the major factors affecting the performance of lazy learning.

Furthermore, we have performed additional experiments on a large set of random instances generated with a range of different parameters and thus confirmed the importance of mean tightness.

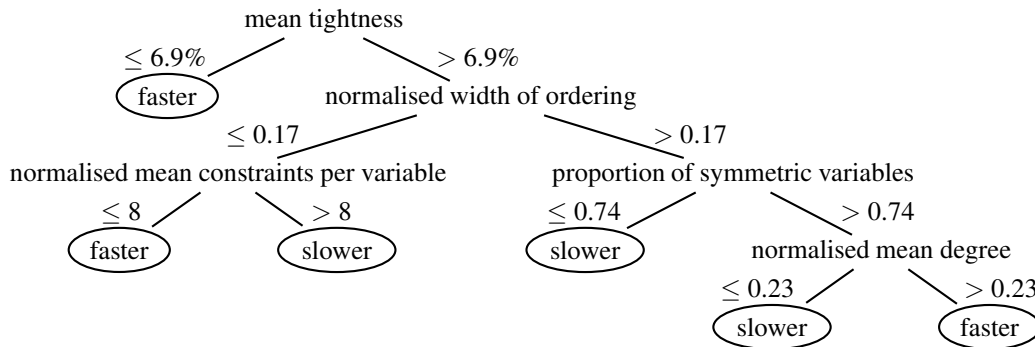


Figure 3: Typical decision tree as determined by the J48 classifier. Starting at the root node, we traverse the tree based on the values of the instance attributes until we reach a leaf that predicts whether lazy learning is faster or slower for this instance.

In the future we would like to investigate the effect of other attributes in conjunction with the mean tightness with the goal of being able to provide a decision algorithm that can be used for any problem instance.

We would also like to extend our study to other components of the constraint solver, in particular ones which, like lazy learning, carry a high overhead, but also provide a large potential gain.

## References

- Ansótegui, C.; Sellmann, M.; and Tierney, K. 2009. A gender-based genetic algorithm for the automatic configuration of algorithms. In *CP*, 142–157.
- Beldiceanu, N.; Carlsson, M.; and Rampon, J.-X. 2005. Global constraint catalog. Technical Report 08, Swedish Institute of Computer Science.
- Borrett, J. E.; Tsang, E. P. K.; and Walsh, N. R. 1996. Adaptive constraint satisfaction: The quickest first principle. In *ECAI*, 160–164.
- Dechter, R. 2003. *Constraint Processing*. Elsevier Science.
- Epstein, S. L.; Freuder, E. C.; Wallace, R. J.; Morozov, A.; and Samuels, B. 2002. The adaptive constraint engine. In *CP*, 525–542.
- Gent, I. P.; Jefferson, C.; and Miguel, I. 2006. Minion: A fast scalable constraint solver. In *ECAI*, 98–102.
- Gent, I. P.; Miguel, I.; and Moore, N. C. 2010. Lazy explanations for constraint propagator. In Carro, M., and Peña, R., eds., *PADL 2010*, LNCS. Springer.
- Guerra, A., and Milano, M. 2004. Learning techniques for automatic algorithm portfolio selection. In *Proc. ECAI 2004*, 475–479.
- Hall, M.; Frank, E.; Holmes, G.; Pfahringer, B.; Reutemann, P.; and Witten, I. 2009. The WEKA data mining software: An update. *SIGKDD Explorations* 11(1).
- Hutter, F.; Hamadi, Y.; Hoos, H. H.; and Leyton-Brown, K. 2006. Performance prediction and automated tuning of randomized and parametric algorithms. In *CP*, 213–228.
- Katsirelos, G., and Bacchus, F. 2003. Unrestricted nogood recording in CSP search. In *CP*, 873–877.
- Katsirelos, G., and Bacchus, F. 2005. Generalized nogoods in CSPs. In *AAAI*, 390–396.
- Katsirelos, G. 2009. *Nogood Processing in CSPs*. Ph.D. Dissertation, University of Toronto. <http://hdl.handle.net/1807/16737>.
- KhudaBukhsh, A. R.; Xu, L.; Hoos, H. H.; and Leyton-Brown, K. 2009. SATenstein: Automatically building local search SAT solvers from components. In *IJCAI*, 517–524.
- Lagoudakis, M. G., and Littman, M. L. 2000. Reinforcement learning for algorithm selection. In *AAAI/IAAI*, 1081.
- Lecoutre, C. 2009. *Constraint Networks*. Wiley.
- Lecoutre, C. 2010. Xcsp benchmarks. <http://www.cril.univ-artois.fr/~lecoutre/benchmarks.html>.
- Leyton-Brown, K.; Nudelman, E.; Andrew, G.; McFadden, J.; and Shoham, Y. 2003. A portfolio approach to algorithm selection. In *IJCAI*, 1542–1542. Morgan Kaufmann.
- McKay, B. 1981. Practical graph isomorphism. In *Numerical mathematics and computing, Proc. 10th Manitoba Conf., Winnipeg/Manitoba 1980, Congr. Numerantium 30*, 45–87. See also <http://cs.anu.edu.au/people/bdm/nauty>.
- Minton, S. 1996. Automatically configuring constraint satisfaction programs: A case study. *Constraints* 1(1/2):7–43.
- Mitchell, T. M. 1997. *Machine Learning*. McGraw-Hill.
- Rice, J. R. 1976. The algorithm selection problem. *Advances in Computers* 15:65–118.
- Various. 2010. SAT competition. [www.satcompetition.org](http://www.satcompetition.org).
- Watts, D., and Strogatz, S. 1998. Collective dynamics of ‘small-world’ networks. *Nature* 393:440–442.
- Xu, L.; Hutter, F.; Hoos, H. H.; and Leyton-Brown, K. 2008. SATzilla: Portfolio-based algorithm selection for SAT. *J. Artif. Intell. Res. (JAIR)* 32:565–606.