USING ALGORITHM SELECTION AND CONFIGURATION ON CONSTRAINT PROPAGATION

by

Daniel J. Geschwender

A THESIS

Presented to the Faculty of

The College of Art and Sciences at the University of Nebraska

In Partial Fulfilment of Requirements

For the Degree of Bachelor of Science

Major: Computer Science

Under the Supervision of Professor Berthe Y. Choueiry

Lincoln, Nebraska

May, 2014

USING ALGORITHM SELECTION AND CONFIGURATION ON CONSTRAINT PROPAGATION

Daniel J. Geschwender, B.S.

University of Nebraska, 2014

Adviser: Berthe Y. Choueiry

This thesis discusses the application of algorithm selection and configuration techniques to the field of Constraint Processing. A wide range of constraint propagation algorithms exist, each with unique strengths and weaknesses. We examine how to make better use of constraint propagation through the proper pairing of problem and algorithm.

Our study is undertaken from two perspectives. First we consider how to select an algorithm for a given problem. We show how a simple machine learning classifier can be constructed to select between two constraint minimality algorithms. Next, we consider building a problem suited to a given algorithm. We demonstrate how an algorithm configurator can be used to guide the construction of problem classes favoring the algorithm. Both approaches are fruitful and have the potential to improve constraint propagation.

ACKNOWLEDGMENTS

This thesis would not have been possible without the outstanding opportunities, mentoring, and support I received from my advisor, Dr. Berthe Y. Choueiry. I want to thank her most of all for getting me to this point.

I have had the opportunity to collaborate with several professors who have been a great help in my work. Dr. Matthew Dwyer, Dr. Stephen Scott, Dr. Holger Hoos, Dr. Frank Hutter, and Dr. Kevin Leyton-Brown have all provided valuable input in this research and for that I would like to thank them.

Dr. Laura Damuth of the University Honors Program has been instrumental in my applications for fellowships to support this research. I want to thank her for her valuable assistance.

I would also like to thank Shant Karakashian, Tony Schneider, and Robert Woodward for their help and suggestions on my experiments.

Finally, I want to thank my friends and family for being patient and supporting me through my long hours of work.

This research was supported by NSF Grant No. RI-111795, Barry M. Goldwater scholarship, and Undergraduate Creative Activities and two grants from the Research Experiences Program (UCARE) of the University of Nebraska-Lincoln. Experiments were conducted on the equipment at the Holland Computing Center at UNL.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Constraint Processing is an expressive and powerful framework for modeling and solving constrained combinatorial problems. Problems from countless real-world applications such as scheduling, resource allocation, and hardware verification can be expressed as Constraint Satisfaction Problems. Solving a Constraint Satisfaction Problem (CSP) is in general NP-complete. The only sound and complete algorithm for solving is exhaustive backtrack search. However, to supplement the search process, there exist a myriad of inference strategies that can simplify a CSP. These constraint propagation algorithms come in many different strengths and enforce varying levels of consistency upon the CSP. Algorithms that achieve a greater amount of filtering may take significantly longer to run. Because of this trade-off, there is rarely a clear-cut best algorithm. An important research direction is the selection of the appropriate algorithms to employ in solving a given problem instance. The inverse question of what problems are particularly suited to the strengths of a given algorithm is similarly interesting. We explore both approaches in this thesis. Investigating these questions has the potential to yield incredibly useful results. By correctly applying propagation algorithms, CSP solvers are able to significantly speed their search for a solution.

Consequently, the Constraint Satisfaction Problem can become another step closer to tractability.

1.1 Contributions

In this thesis we address both the question of what algorithm to select for a given problem and the question of what type of problem is best suited to a given algorithm. In answering these questions, we make use of two algorithms for enforcing a high level of consistency: constraint minimality. The two algorithms, PerTuple and AllSol, enforce this consistency in different ways and thus have differing performance. We make the following contributions:

1. We demonstrate and compare several machine learning classifiers that can be used to select when to use the two algorithms.

2. We highlight a class imbalance in the problems favoring PerTuple and the problems favoring AllSol.

3. We demonstrate a framework for configuring randomly generated CSP instances to favor one algorithm over another.

4. We show what problem features cause PerTuple to run faster than AllSol and vice-versa.

1.2 Outline of Thesis

The thesis is structured as follows:

Chapter 2 In this chapter, we present background material concerning the Constraint Satisfaction Problem. It also addresses background in algorithm se-

lection and configuration. We give a synopsis of papers dealing with related research.

Chapter 3 In this chapter, we discuss the constraint minimality property and the algorithms PerTuple and AllSol. We compare various classification strategies and identify one that performs the most accurate classification. At the end, we show the distribution of AllSol and PerTuple problems and discuss how this affects the classification.

Chapter 4 This chapter covers the question of which types of problems suit a given algorithm. We show how to use both a random CSP generator and an algorithm configurator to automatically create a problem built for use with a particular algorithm. We give details on the effectiveness of such an approach and look at what parameters make a problem better suited to AllSol or PerTuple.

Chapter 5 This last chapter concludes this thesis, and gives our ideas for future research.

Chapter 2

Background

2.1 The Constraint Satisfaction Problem

A Constraint Satisfaction Problem (CSP) is defined by the tuple $\mathcal{P} = \{\mathcal{V}, \mathcal{D}, \mathcal{C}\}$:

1. $\mathcal{V} = \{V_1, V_2, ..., V_n\}$ is a set of variables.
2. $\mathcal{D} = \{D_1, D_2, ..., D_n\}$ is a set of finite domains. Each variable $V_i \in \mathcal{V}$ has a corresponding domain $D_i \in \mathcal{D}$
3. $\mathcal{C} = \{C_1, C_2, ..., C_e\}$ is a set of constraints. Each constraint restricts the possible assignments of values to variables.

Each constraint $C_i \in \mathcal{C}$ is defined by a *scope* and *relation*. The $scope(C_i) \subseteq \mathcal{V}$ is the subset of variables in $\mathcal{V}$ that the constraint applies to. The *arity* of a constraint refers to the magnitude of the constraint's scope, i.e., $arity(C_i) = |scope(C_i)|$. A constraint with arity 1 is known as a *unary* constraint while those with arity 2 are *binary*. Constraints covering all n variables are *global*. A CSP containing only unary and binary constraints is a binary CSP. The $rel(C_i) \subseteq D_x \times D_y \times ... \times D_z$ is the set of allowed tuples of elements from the scope of C_i.

The *density* of a CSP is a measure of the number of constraints in the problem. It is equal to $\frac{e}{e_{max}}$ where e is the number of constraints and $e_{max} = \frac{n(n-1)}{2}$. The *tightness* of a CSP is a measure of the restrictiveness of the constraints in the problem. It is equal to $\frac{|forbiddentuples|}{|alltuples|}$. The *phase transition* is a phenomena witnessed when varying the tightness of problems. A low tightness will cause solutions to be plentiful and easy to find. A high tightness will make the problem over constrained and thus it will be easy to determine that no solutions exit. However, in the intermediate regions of tightness finding solutions becomes increasingly difficult.

As an example, we will show how a map coloring problem can be represented as a CSP. The map will be made up of four regions: Nebraska (NE), Iowa (IA), Kansas (KS), and Missouri (MO). They are laid out as shown in Figure 2.1. Each region can be colored one of three different colors: red, green, or blue. There is a restriction that any two adjacent regions may not share a color. A solution is a coloring of the regions that obeys this restriction.

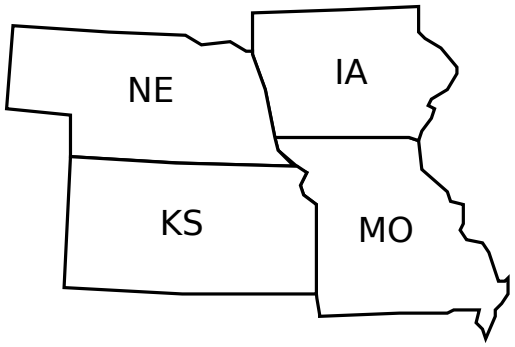

Figure 2.1: An example map coloring problem.

To model this problem as a Constraint Processing Problem, we need to identify variables, domains, and constraints. The variables will be the map regions. Thus $\mathcal{V} = \{NE, IA, KS, MO\}$. The domains will be the colors that each region can take. So $\mathcal{D} = \{D_{NE}, D_{IA}, D_{KS}, D_{MO}\}$ and $D_{NE} = D_{IA} = D_{KS} = D_{MO} = \{r, g, b\}$. The

constraints are binary 'not-equal' constraints between all adjacent regions. Thus $C_{NE,IA} = C_{NE,KS} = C_{NE,MO} = C_{IA,MO} = C_{KS,MO} = \{(r,g),(r,b),(g,r),(g,b),(b,r),(b,g)\}$. This CSP is illustrated in Figure 2.2.

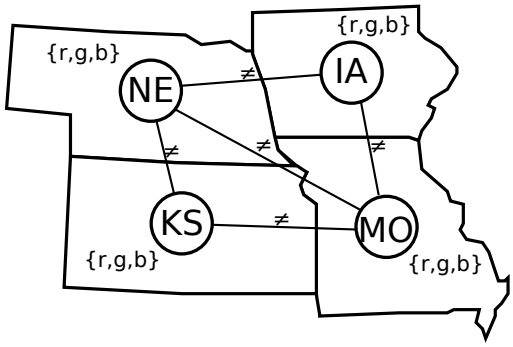

Figure 2.2: Map coloring CSP.

A solution to a CSP is an assignment of values to variables. The values are taken from the respective domains of the variables and must satisfy all constraints. When given a CSP, it may be desirable to find a single solution, enumerate all possible solutions, or determine if any solutions exist.

In our map coloring example, one solution would be: $NE = r, IA = g, KS = g, MO = b$. Upon examination, we can see that KS and IA must always be colored the same. The remaining two colors are used on NE and MO. Thus if we were to enumerate all solutions, we would find 6 unique colorings of the map. Trivially, we can see that a solution exists if we can enumerate 6 solutions. However, in some CSPs, no solution exists and it may be difficult to determine short of an exhaustive search.

There are many techniques and algorithms utilized in solving a CSP. Generally these can be classified into two categories: *search* and *inference*.

2.1.1 Search

Search is the process of exhaustively enumerating all possible variable-value combinations for a given CSP in order to find a solution. The most basic search technique, *backtrack search*, involves systematically assigning values to variables until a conflict is reached or a solution discovered. Upon finding a conflicting assignment, the value is thrown out and the next domain value is used. When all values in a domain have been tried, *domain wipeout* occurs and search backtracks to reassign the previous variable. In this way, the entire search space is explored in a depth first manner.

There are many variations upon the simple backtrack search. The ordering in which the variables are assigned values can have a significant impact on the search. For instance, if many conflicts are detected early and high in the search tree, entire branches of the search may be avoided. Similarly, the ordering in which values are taken from the variable domains can have an impact. A large number of heuristics exist for determining orderings of both variables and values.

Backtrack search can be enhanced by look-ahead strategies such as *forward checking*. A forward checking search will examine all values in unassigned variables to see if they conflict with currently assigned values. Thus it looks forward and checks the upcoming values. Conflicting values are removed. This may lead to early domain wipeouts and prevent unnecessary branches from being explored. Forward checking is a simple inference technique, though there are many that are more complex.

2.1.2 Inference

Inference is a technique that utilizes information about the constraints to reduce a CSP to a simpler problem. While inference is not complete and thus not guaranteed to find a solution, it often can drastically speed the solution search. Inference takes the

form of *constraint propagation algorithms*, which are used to enforce *local consistency* properties on CSP.

A consistency property defines to what extent a CSP is internally consistent. By enforcing such a property, domain values that do not meet this consistency requirement may be filtered out and the resulting problem simplified. Depending on the strength of the consistency property, more or less filtering may occur in differing amounts of computation time.

The simplest and most used consistency property is *arc consistency*. A value is arc consistent if it has a *support*, a neighboring value with which it is consistent. A variable is arc consistent if all its values are. Similarly, a CSP is considered to be arc consistent if all variables are arc consistent. Thus every value in the CSP must have a support. There are numerous propagation algorithms for enforcing arc consistency. The standard AC algorithm operates on binary CSPs. The Generalized Arc Consistency (GAC) algorithm operates on constraints of any arity.

There are many more complex uses of inference. Path consistency requires a chain of supports of a given length for every value. Maintaining Arc Consistency (MAC) interleaves search and propagation to have an arc consistent CSP at every step. Relational consistencies operate on the constraints rather than the domains. Algorithms that enforce relational consistencies filter out tuples from the relations.

2.2 Algorithm Selection

Algorithm selection refers to any method used to choose an algorithm for a given problem. The question of how to determine the optimal algorithm to apply in a given situation was addressed in the paper "The Algorithm Selection Problem" by John Rice [1976]. This paper discusses algorithm selection in a general sense rather than in the

context of a particular domain. Rice presents four steps that apply to all algorithm selection problems: the extraction of key features of the problem, the mapping of those features to a selected algorithm, the mapping of the algorithm to performance measures, and a norm on the performance measures to evaluate the algorithm. An *algorithm portfolio* is a collection of algorithms and a policy for selecting one or more of the algorithms to run [Huberman *et al.*, 1997]. Ideally the constituent algorithms will complement each other and make up for each other's failings. Algorithm portfolios have found great success in the related field of Boolean satisfiability with the SATzilla SAT solver [Xu *et al.*, 2008].

2.3 Algorithm Configuration

Algorithm configuration is an optimization technique used to improve the performance of a target algorithm on a particular set of instances by tuning the algorithm's input parameters. Algorithm configuration is typically used to reduce the runtime of the target algorithm, and it can also be used to improve the algorithm's solution quality. Various configuration strategies are used to optimize an algorithm, including racing procedures [Birattari *et al.*, 2002], local search [Hutter *et al.*, 2009], and model-based optimization [Hutter *et al.*, 2011]. Algorithm configuration is most effective when as much as possible of the internals of the algorithm are exposed as parameters, increasing the flexibility of the configuration. Algorithm configuration is particularly useful in that it allows for detailed automated exploration of the parameter space and can eliminate the need for manual tuning by a domain expert.

2.4 Related Work

Finding the best algorithm to handle a given problem is an important endeavor. This is an active area of research and thus there is much work related to our own. The following are papers most closely related to our own.

Chmeiss and Sais [2004] examine ways to enforce consistency during backtrack search while avoiding the cost of running full MAC. The paper introduces MAC(dist), an algorithm that enforces AC at every step of the search out to a distance specified by the parameter. This allows the algorithm to perform anywhere on a continuum between FC and MAC. This intermediate consistency is useful on the class of sparse random CSPs tested on.

Stergiou [2009] identifies 4 heuristics to aid in the selection of a strong consistency or weak consistency. In this paper, AC is used as the weak consistency while maxRPC is used as the strong. The heuristics operate by keeping track of value deletions and domain wipeouts. When the heuristic value passes some threshold, strong consistency will be applied. Later, it may return to the weak consistency if the strong is no longer necessary. Utilizing the heuristics resulted in time savings during search, particularly when combining multiple heuristics.

Epstein *et al.* [2005] develop 'propagation policies,' strategies specifying what algorithms to apply when during the search. The ACE system used in the experiments is given several consistency algorithms and run iteratively over a set of CSP instances, updating the policy as it goes. In this study, six (6) algorithms were used, all variants of FC and AC. These variants are based around limiting the neighborhood of allowed propagation, quantifying the responsiveness of a filtering operation, and accounting for the current depth of the search. Experiments show ACE develops close to an ideal policy for the algorithms it is given.

Balafrej *et al.* [2013] define a general form of parameterized consistency that can scale in strength from AC to any stronger consistency desired. The scaling is based around the 'p-stability' of the values, which is determined by the distance of the value from the end of the ordering of values. The paper uses maxRPC as the strong consistency and thus p-maxRPC is the parameterized consistency. Adaptive forms of this consistency are also introduced, which change the parameter value in response to conditions in the search. Tests show the adaptive consistency outperforms the basic strong and weak consistencies.

Woodward *et al.* [2011] make use of a machine learning classifier to select between the use of four (4) variants of a consistency algorithm. The base consistency used is Relational Neighborhood Inverse Consistency. There is a variant that first removes redundant edges, a variant that first triangulates the dual graph, and a variant that does both. A simple hand-tuned decision tree selects the algorithm based on features of the CSP. This selection process results in better performance than any of the individual algorithms.

Chapter 3

Machine Learning Classifier to Select a Minimality Algorithm

A major limiting factor in the solving of CSPs is the time complexity. By their nature, the combinatorial problems dealt with in Constraint Processing are difficult to solve. Luckily, in practice, it is often possible to solve a CSP in under exponential time, given the right algorithm. Constraint network minimization is a technique that can be used to simplify a CSP by removing unnecessary tuples from the constraints [Montanari, 1974]. In this chapter, we consider two minimization algorithms: PerTuple and AllSol. Depending on the CSP being minimized, the performance of two algorithms can vary widely, to the point where one or the other may not terminate. Therefore it is of great importance to select the correct algorithm. We show how to use machine learning classifiers to accomplish this task.

3.1 Enforcing Constraint Minimality

Constraint minimality is a strong consistency property that operates on the relations of the CSP. It guarantees that every tuple in every constraint's relation appears in a solution to the CSP [Montanari, 1974], see Figure 3.1. The importance of minimality was established for knowledge compilation [Gottlob, 2012] and achieving higher levels of consistency [Karakashian *et al.*, 2013].

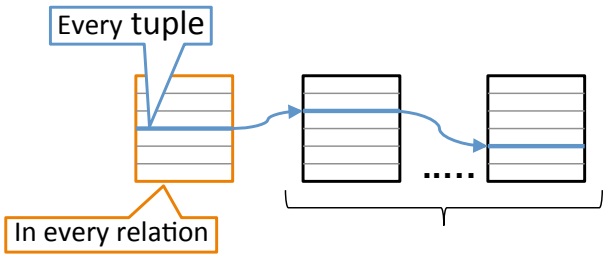

Figure 3.1: Minimal relations.

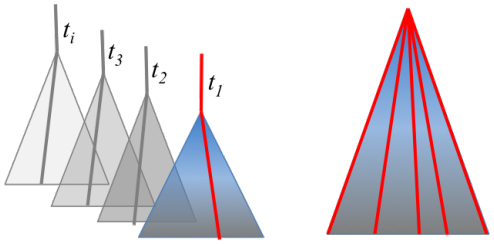

Figure 3.2: Minimality algorithms: PerTuple & AllSol.

3.1.1 PerTuple

PerTuple loops over all tuples of every relation of the CSP. For each tuple, it performs a backtrack search to find the first solution in which the tuple appears. If no solution is found involving the tuple, the tuple is removed from the relation. If a solution is found, all tuples involved in the solution are saved. PerTuple terminates after having removed or saved every tuple. The number of search processes executed by PerTuple is linear in the total number of tuples in the relations. Intuitively, PerTuple should perform well when each search can be quickly completed, that is, instances that are small or instances that are large but located away from the phase transition.

Algorithm 1: PERTUPLE($\mathcal{P}_D$) [Karakashian *et al.*, 2012]

Input: $\mathcal{P}_D$
Output: Minimal Network of $\mathcal{P}_D$

```
1 foreach R_i ∈ P_D do
2     foreach τ_i ∈ R_i do SETMARK(τ_i, false)
3 foreach R_i ∈ P_D do
4     foreach τ_i ∈ R_i, do
5         if MARKED(τ_i) = false then
6             ASSIGN(R_i, τ_i)
              /* Backtrack search for a solution                    */
7             sol ←BTSEARCHONESOL(P_D)
8             if sol = false then DELETE(τ_i)
9             else   foreach τ_j ∈ sol do SETMARK(τ_j, true);
```

3.1.2 AllSol

To enforce minimality, the AllSol algorithm executes a single backtrack search that enumerates all the solutions. For every solution identified, the involved tuples are saved. AllSol terminates after exploring the entire search tree. Tuples not appearing in a solution are removed. AllSol only performs a single but exhaustive backtrack search. Thus, AllSol should outperform PerTuple in large problems that are around or above the phase transition where most of the search space is explored anyway. In problems with plentiful solutions, AllSol may become overwhelmed by enumerating all the solutions. The situation is depicted in Figure 3.2.

3.2 Problem Features

In order to build a machine learning classifier, features need to be selected in order to characterize the objects being classified. We use 12 features that provide defining information about a given CSP. These features can be calculated in relatively short time by examining the structure and constraints of the problem. The 12 features are:

Algorithm 2: AllSol($\mathcal{P}_D$) [Karakashian *et al.*, 2012]

Input: $\mathcal{P}_D$
Output: Minimal Network of $\mathcal{P}_D$

```
1  foreach R_i ∈ P_D do
2      foreach τ_i ∈ R_i do SetMark(τ_i, false)
3  sol ← false
4  while sol = false do
5      sol ←BTsearchNextSol(P_D)
6      if sol ≠ false then
7          foreach τ_i ∈ sol do SetMark(τ_i, true)
8  foreach R_i ∈ P_D do
9      foreach τ_i ∈ R_i do
10         if Marked(τ_i) = false then Delete(τ_i)
```

1. κ
2. $log_2(avg(\text{relLinkage}))$
3. $log_2(stDev(\text{relLinkage}))$
4. $stDev(\text{relLinkage})/avg(\text{relLinkage})$
5. $stDev(\text{tupPerVvp})/avg(\text{tupPerVvp})$
6. $avg(\text{tupPerVvpNorm})$
7. $stDev(\text{tupPerVvpNorm})$
8. $stDev(\text{tupPerVvpNormProd})$
9. $stDev(\text{tupPerVvpNormProd})/avg(\text{tupPerVvpNormProd})$
10. $avg(\text{relPerVar})$
11. $stDev(\text{relPerVar})$
12. $stDev(\text{relPerVar})/avg(\text{relPerVar})$

κ is a parameter used to predict if the instance is at the phase transition [Gent *et al.*, 1996]. *relLinkage* measures the likelihood of a tuple at the overlap of relations being in a solution. *tupPerVvp* counts the number of tuples a variable-value pair appears in. *relPerVar* counts how many relations there are per variable. These values are combined with various statistical aggregations to obtain the 12 features.

3.3 Building the Classifier

In order to build a strong classifier, a large data set must be used to train it. We used benchmarks problems from the 2008 CSP Solver Competition. Each benchmark is a set of CSP problems coming from a given problem domain and having similar structure. By drawing upon many benchmarks, the classifier will be more generalized and avoid fitting to a specific type of problem.

Each instance in each benchmark is broken down into clusters using the CSP tree decomposition. Each cluster is a smaller subproblem that can treated as a complete CSP and filtered using either algorithm. The tree decomposition is performed because it corresponds to one intended use of the two algorithms [Karakashian *et al.*, 2013]. This decomposition also serves to provide smaller problems to handle as well as a larger data set. AllSol and PerTuple are both run on every cluster of the decomposed CSP. Execution time of each algorithm, as well as all 12 problem features are output for every cluster.

From the features and execution times output, we compile a dataset. Each instance is assigned to a class of either AllSol or PerTuple, depending on which executed faster. Each instance is also given 12 attributes corresponding to the 12 features extracted from the problem. This data set is then run through one of the algorithms of the Weka machine learning suite [Hall *et al.*, 2009] and evaluated based on how well

it classifies the instances.

3.4 Experiments

In order to determine the feasibility and effectiveness of this algorithm selection approach, we ran several experiments on a set of data collected from five sets of benchmark problems: aim50, aim100, aim200, renault, and warehouse. This resulted 62 CSPs and 3592 total data instances.

The data presented in Table 3.1 shows some general statistics about the five benchmarks individually as well as combined. This data shows that there is in fact a noticeable difference in algorithm performance between the benchmarks. The class distribution varies widely between the sets: a fairly even 467 to 517 for the renault benchmark compared to the one sided 201 to 1 for the warehouse benchmark. The difference in mean execution times also varies depending on the benchmark. The aim100 benchmark has a substantially longer AllSol execution time. The warehouse benchmark, though heavily siding towards the AllSol algorithm, had comparable execution times for each. This indicates that the execution time difference was only slightly in favor of AllSol.

Table 3.1: Benchmark Statistics.

Benchmark	**# AllSol**	**# Per-Tuple**	**Mean AllSol Time** (ms)	**StdDev AllSol Time** (ms)	**Mean PerTuple Time** (ms)	**StdDev PerTuple Time** (ms)
aim50	130	444	352.65	2,890.86	68.75	201.90
aim100	158	747	16,584.45	474,935.80	55.09	49.66
aim200	174	753	264.76	3,711.17	140.38	72.94
renault	467	517	2,151.40	605.65	2,180.84	684.58
warehouse	201	1	628.76	347.07	724.21	476.07
(combined)	1130	2,462	4,927.83	238,499.30	699.25	999.84

As a first experiment, each benchmark data set, as well as the combined data set,

was run through three Weka algorithms to generate classifiers. The three classifiers chosen were: J48, MultilayerPerceptron, and NaiveBayes. During the training we used 10-fold cross-validation. The data in Table 3.2 shows the F-measures of each data set with each classifier. F-measure is a statistical analysis of classification and is calculated from the harmonic mean of precision and recall. An F-measure of 1 represents perfect classification. For the most part, the three classifiers all performed similarly. The F-measure differed mainly between benchmarks. Some benchmarks are easier to classify than others. The Warehouse benchmark was able to achieve almost perfect classification, though this is due to the fact that the classes were split 201 to 1. The renault benchmark had the lowest F-measures but they were still above random guessing. The F-measure of the combined data was about 0.73 for all classifiers. This represents a reasonably accurate classification, though it could still see some improvement.

Table 3.2: Benchmark F-measures.

Benchmark	**J48**	**MultilayerPerceptron**	**NaiveBayes**
aim50	0.675	**0.689**	0.463
aim100	**0.746**	0.745	0.745
aim200	**0.729**	0.720	0.718
renault	0.551	0.578	**0.589**
warehouse	**0.993**	**0.993**	**0.993**
(combined)	0.726	0.726	**0.728**

In an attempt to increase the classifier F-measure, some modifications were made to the data set and classifier training was run again. These tests were only run on the combined data from all benchmarks. 10-fold cross-validation was again used. Table 3.3 shows the results of these tests. The first row shows the F-measures on the basic data set again. The next test cut out a portion of the data. The instances where the execution times were within 100 ms of each other were dropped out. The

F-measure rose significantly as a result. This indicates that the instances having similar performance on both algorithms are difficult to classify. The next row shows the results of splitting into three classes. Instead of throwing out instances in which the difference is less than 100 ms, a third class, Either, is created. The F-measures with this data set are higher still. However, this result is somewhat misleading. The bulk of the instances actually fall into the Either class. This class is easy to classify properly but almost meaningless as it will result in no time savings. The classes we really care about predicting accurately are AllSol and PerTuple. The fourth row shows the F-measure on the AllSol and PerTuple instances with classifiers trained on all three classes. The results are poor and not an improvement over throwing out the close instances. These tests show that it is possible to improve the F-measure by modifying the data set.

Table 3.3: Weighted average F-measure of the three algorithms.

Strategy	**J48**	**MP**	**NB**
All instances	0.726	0.726	0.728
$\delta_t \geq$100ms	**0.917**	0.880	0.900
3 classes	0.936	0.941	0.871
PerTuple+AllSol	0.501	0.547	0.433

Throwing out the data instances with minor execution time differences helped to emphasize the more important data instances. To further emphasize the most meaningful data instances, we experimented with weighted data sets and cost sensitive classifiers on the J48 classifier. 10-fold cross-validation was incompatible with the weighting in Weka and thus was not used. In the weighted data set there are still two classes: AllSol and PerTuple. Each instance is also given a weight equal to the difference in execution times of the algorithms. Therefore, an instance with a difference of 100000 ms is given considerably more importance than an instance with a difference of 10 ms. We also created a cost matrix for cost sensitive classification [Elkan, 2001].

Classifying either class properly has a cost of 0. Classifying an AllSol instance as PerTuple has an average cost of 59 ms. Classifying a PerTuple instance as AllSol has an average cost of 6196 ms. Thus the classifier will err towards PerTuple, as the cost is less.

First, we trained on the unmodified data set and obtained weighted and unweighted F-measures and time savings information. The time saved and lost are the actual amounts of time saved and how much more could have potentially been saved. We also report the percentage of the total possible time savings that were realized.

The resulting F-measures for J48 are shown in Table 3.4. The accuracy on the weighted set gives the percent of potential time savings that was actually obtained. While not achieving perfect F-measures, all four classifiers saved over 99% of the time possible to save. All the significant instances were properly classified, only the more trivial instances were incorrect. The classifier trained on the weighted dataset marginally achieved the best F-measure and time savings. The classifier trained with the cost matrix had the worst performance. Indeed, the cost matrix takes into account the average cost of a misclassification, however, the standard deviation of the execution time is so high that the average is not particularly relevant. We conclude that J48 with the weighted dataset seems to be the most promising.

Table 3.4: Performance of J48 on four strategies.

Strategy	**F-measure**	**Time saved**		**Time lost**
		%	ms	ms
All instances	0.727	99.87%	15,301,950	19,350
$\delta_t \geq$100ms	0.729	99.90%	15,306,510	14,790
Weighted	**0.743**	**99.96%**	**15,314,980**	**6,320**
Cost	0.557	99.57%	15,255,190	66,110

Figure 3.3 shows the distribution of the instances and the effect of the classification with J48 and the weighted dataset. The vertical axis shows the number of seconds

to execute AllSol while the horizontal shows execution time for PerTuple. Any data point above the diagonal is better suited for PerTuple while anything below is suited to AllSol. The further from the diagonal, the more suited the instance is. We can indeed see that all important instances are classified. Those that are missed are too close to the diagonal to have a significant impact. We also see a clear imbalance in the classes. There are no instances from the five benchmarks that significantly favor AllSol.

After noticing this class imbalance, we collected algorithm runtime data over a much larger set of benchmarks. We took 86 benchmarks from the 2008 CSP Solver competition. Figure 3.4 shows the distribution of runtimes. While we do see a handful of more example favoring AllSol, substantially more favor PerTuple. This imbalance may be representative of the distribution of all CSPs, but it can pose a problem to training a classifier. More examples of problems favoring AllSol are needed to avoid building a biased classifier. In Chapter 4 we demonstrate how to find these problems.

3.5 Summary

Minimizing the constraint network can be a beneficial step in solving a CSP. Unfortunately, using the wrong minimality-enforcing algorithm can cost a substantial ammount execution time. By using machine learning to look at key features of a problem and predict the optimal algorithm to apply, a great deal of time can potentially be saved. The experimental results presented in this chapter show that a machine learning approach is feasible and promising. However, we also highlight an imbalance in the number of problems on which PerTuple runs faster and number of problems on which AllSol runs faster. To correct for potential biasing of the classifier, we need to train with a larger number of AllSol favoring instances.

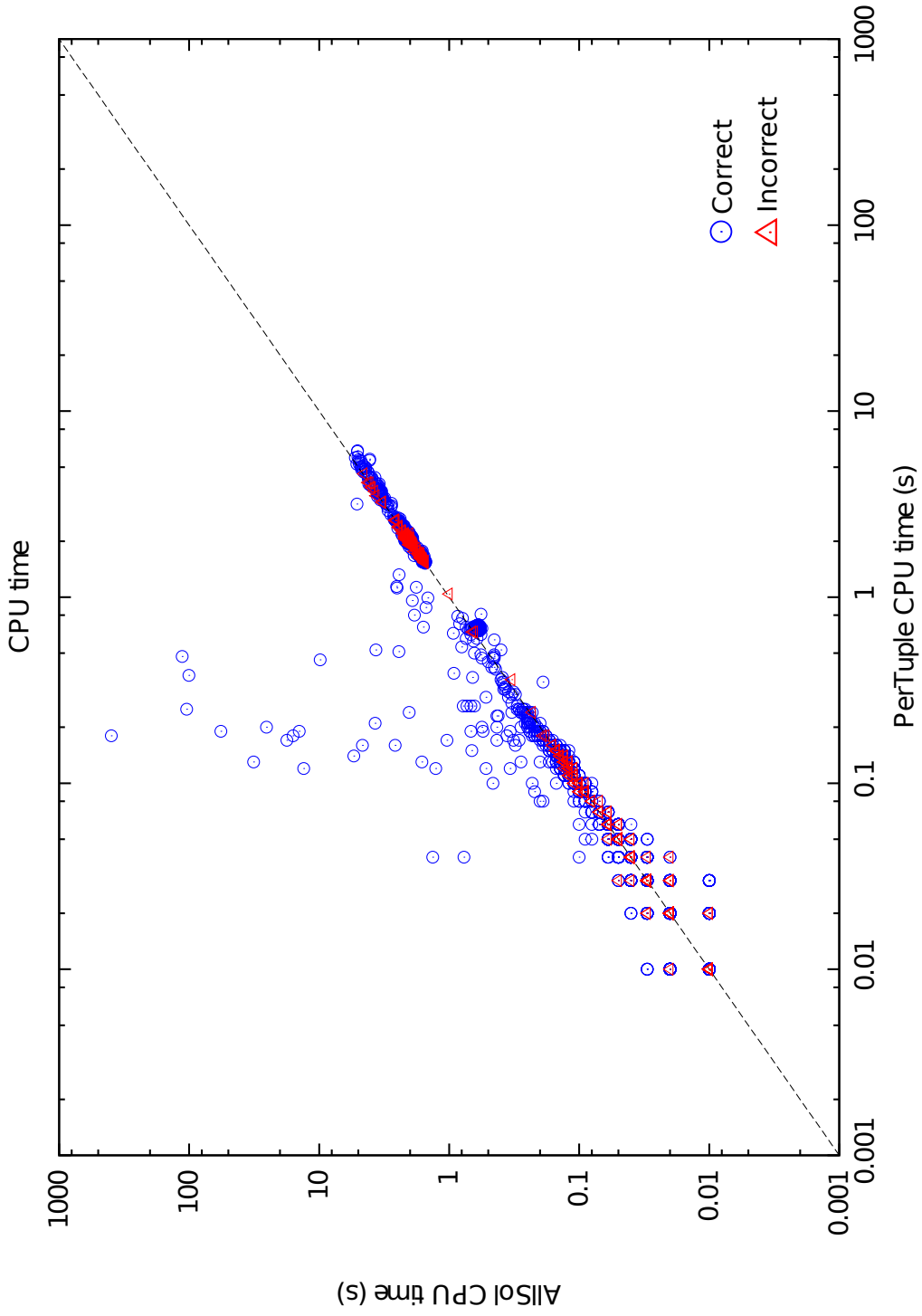

Figure 3.3: Classified instances.

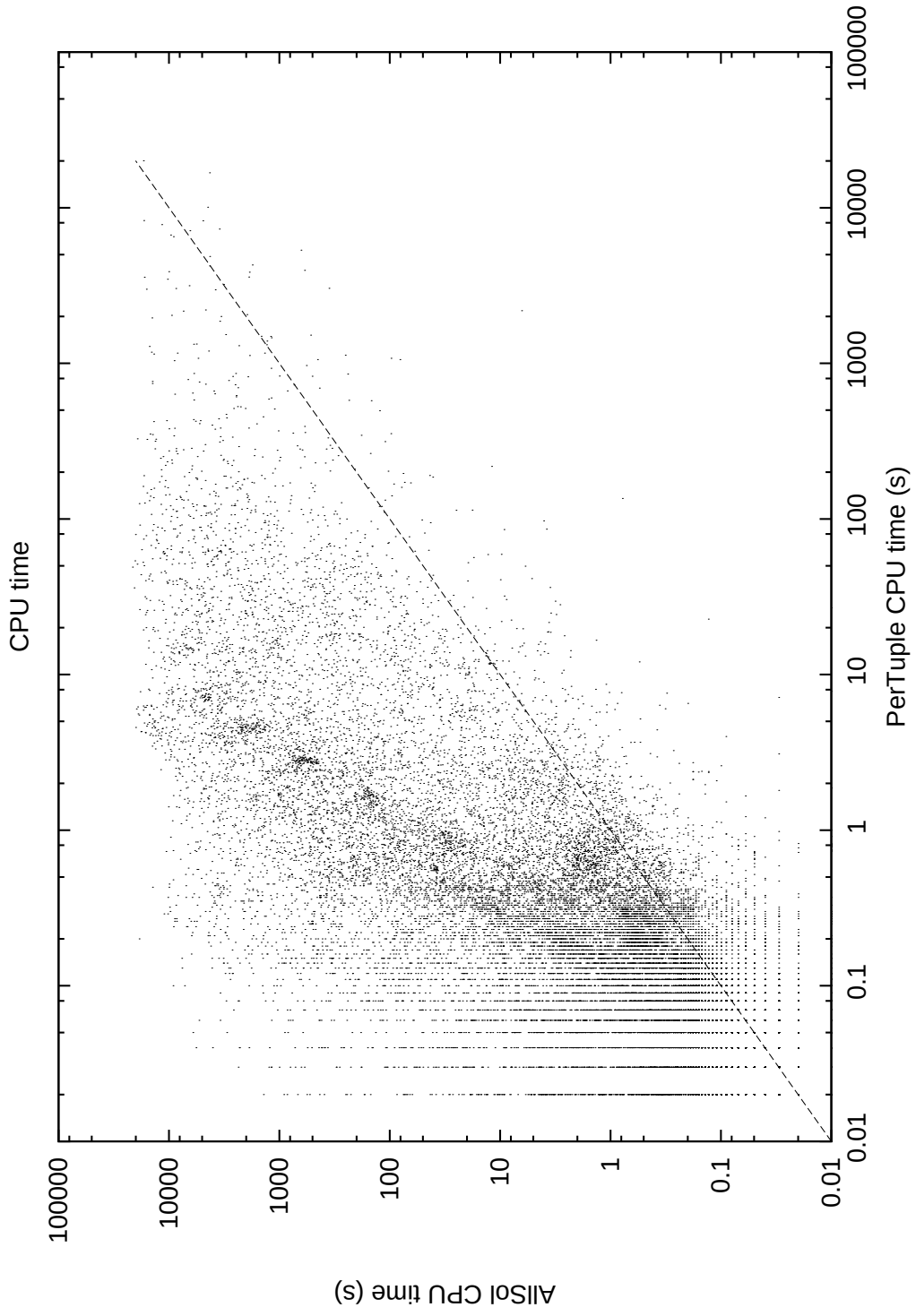

Figure 3.4: Entire set of time data.

Chapter 4

Algorithm Configuration to Guide Random CSP Generation

In this chapter, we demonstrate the utility of algorithm configuration in the random generation of CSP instances. By taking a suitably parameterized random CSP generator, we use an algorithm configurator to tune the parameters such that a given algorithm performs favorably on the generated instance. With this technique we generated instances that run over 1000 times faster on one propagation algorithm than another. We also see what parameters are responsible for this performance difference.

4.1 Configuration of RBGenerator

We use an algorithm configurator that guides a random CSP generator, which generates instances on which we execute PerTuple and AllSol to test their performances. After comparing their performances on the generated instances, the configurator selects new parameters for the CSP generator in order to influence the performances. Figure 4.1 shows the various components of the configuration system.

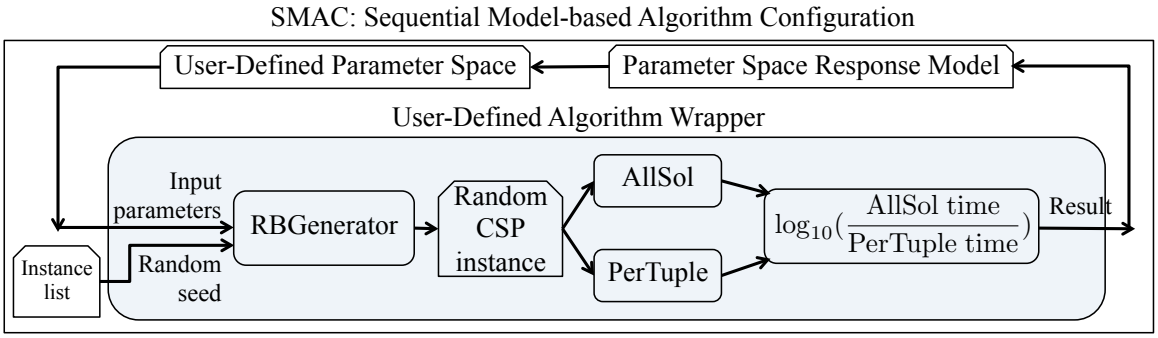

Figure 4.1: Operation of the configurator.

4.1.1 RBGenerator

We use the random CSP generator RBGenerator [Xu *et al.*, 2007]. This generator is based on the model RB, which allows for easy generation of hard satisfiable instances at the phase transition. RBGenerator uses the following parameters:

1. $k \geq 2$ denotes the arity of the constraints
2. n denotes the number of variables
3. α determines the domain size $d = n^{\alpha}$ of each variable
4. r determines the number $m = r \cdot n \cdot \ln(n)$ of constraints
5. δ determines the distance from the phase transition $p_{cr} + \frac{\delta}{1000}$, where $p_{cr} = 1 - e^{-\frac{\alpha}{r}}$
6. *forced* indicates whether or not instances are forced satisfiable
7. *merged* indicates whether or not constraints of similar scopes are joined

A strength of the RBGenerator is that it guarantees an asymptotic phase transition under certain parameter conditions. When an asymptotic phase transition exists, the threshold can be exactly determined. In addition, it provides a useful selection of parameters for the configuration process.

4.1.2 Sequential Model-based Algorithm Configuration

To tune the parameters fed into RBGenerator, we use of the algorithm configurator SMAC (Sequential Model-based Algorithm Configuration) [Hutter *et al.*, 2011]. We give SMAC a description of the input parameters and acceptable ranges for them as well as a default parameter configuration. We also give it a list of instances to use for configuration. In this case, the list of instances is a set of 30 random seeds for the RBGenerator. It is important to note that SMAC also makes use of a configuration seed, which is separate from the RBGenerator instance seed. Finally, we give SMAC a custom algorithm wrapper that handles the execution of RBGenerator, PerTuple, and AllSol. SMAC takes an initial default configuration, performs an algorithm execution, and determines its performance based on the wrapper output. Then, it iteratively repeats the process, selecting new configurations and evaluating them. Configurations are selected based on a continually developing regression model of the parameters. The parameter exploration is also tied to the random configuration seed that is provided to SMAC on launch.

4.1.3 Algorithm Wrapper

The algorithm wrapper encapsulates several programs to be run together. Initially, RBGenerator runs with the parameters provided by SMAC. This generates a CSP instance on which the two consistency algorithms for enforcing minimality are executed. The wrapper is set to handle any crashes or timeouts from it's components. On a successful run, the execution times of PerTuple and AllSol are recorded. These values are compared by taking the base-10 logarithm of their ratio. The numerator and denominator of the ratio determines which algorithm is being optimized for. Taking the logarithm ensures that equal weights are given to fractional values when

the results are averaged in SMAC's model.

4.2 Experiment Setup

In our experiments, we evaluate how well SMAC is able to generate instances that favor a given algorithm. To this end, we test two cases: those where SMAC is allowed to adjust all parameters (denoted adjustable size), and where SMAC has a restricted set of parameters (denoted fixed size). For the restricted set of parameters, we fix n to 16 and α to 1, allowing SMAC to only control the constraints and thus restricting the generated CSPs to be of a fixed size. For each of these two cases, we generate instances favoring PerTuple and instances favoring AllSol (for a total of four tests). Each test is ran 10 times (for each configuration seed of SMAC) resulting in different paths through the parameter space and a better picture of the effects of the configuration.

To prevent the algorithm wrapper from stalling during configuration, we set time limits on its components. RBGenerator is allowed to run for five minute to generate a CSP instance. PerTuple and AllSol are allowed to run for 20 minutes each while they enforce minimality on the CSP. If RBGenerator exceeds its time limit, the entire run is considered crashed. If PerTuple and AllSol exceed their time limit, their runtime is reported as 20 minutes. Restricting the time limit allows for comparison between runs that terminate and those that do not. SMAC is allowed to run configuration runs for four days. After that point, it takes the best configuration and validates the results by running on all 30 RBGenerator instance seeds.

We run all our tests on a computer cluster of 7232 Intel Xeon cores in 452 nodes. Each configuration run was allocated a single core of an Intel Xeon E5-2670 2.60GHz processors and given 3 GB memory.

4.3 Results

In all four cases, the configurator is able to find parameter settings that cause the desired algorithm to significantly outperform the other. Table 4.1 shows the results of the configuration in each of the four tests, across all 10 seeds. The column iters. reports the number of iterations of SMAC before stopping (four days). The reported speedup for each seed is the average speedup across 30 different instances generated with 30 different instance seeds. We also provide the *coefficient of variation* (CoV), which is the ratio of the standard deviation to the mean. A CoV value of less than 50% indicates that the variance across instance seeds is low. Consequently, the results are more dependent on the parameter configuration than on the random variation between the 30 instances.

The maximum speedup found is bolded for each of the four tests. All four tests realize a speedup of at least 100 times, enough to definitively show there are classes of problems heavily suited to either algorithms. However, when configuring for PerTuple, we achieve speedups of over 1000 times. This fact may indicate that PerTuple has a stronger affinity for a particular problem class than AllSol. It is also worth noting that the fixed and adjustable problem size causes little change in the achieved speedup. Adjustable problem size parameters allow discovery of only marginally better configurations.

Note that some of the configuration seeds led to configurations with no speedup, either because of crashes or timeouts. This lack of progress is the result of particular seeds yielding flawed initial parameters. By continuing to select parameters causing crashes and timeouts, SMAC has no useful data on which to build its model. On those seeds, SMAC continues to blindly select new parameters that cause crashes and timeouts.

Table 4.1: Configured parameters and the resulting speedups.

		seed	iters	k	n	α	r	δ	*forced*	*merged*	speedup	CoV
Adjustable Size	**AllSol**	1	4330	4	19	0.75	9.91	578	n	y	101.36	14.55%
		2	2434	7	18	0.20	5.88	-9	y	y	36.30	6.49%
		3	2722	5	17	0.68	9.18	759	n	y	75.73	8.10%
		4	1451	2	20	1.74	8.28	309	n	y	**348.43**	4.83%
		5	393	2	12	1.45	4.43	-54	n	n	11.21	32.69%
		6	270	3	14	0.78	1.52	-100	y	n	9.63	27.00%
		† 7	599	2	20	1.75	8.12	302	y	n	289.63	27.61%
		8	454	3	8	1.70	1.00	-155	n	n	13.72	77.29%
		9	1864	5	19	0.72	7.70	825	n	n	80.70	3.56%
		10	2712	4	16	0.86	9.95	646	n	y	115.01	17.31%
	PerTuple	1	248	2	9	1.28	5.29	-162	n	n	378.29	15.12%
		† 2	562	2	17	0.79	0.61	-394	y	n	**4,627.35**	78.67%
		3	202	3	16	1.01	0.29	-190	n	n	296.94	102.49%
		4	88	2	14	0.77	2.69	-235	y	n	531.85	15.82%
		5	905	2	12	0.85	6.19	-98	y	y	442.20	12.32%
		‡ 6	305	4	13	9.80	8.05	542	n	n	1.00	0.00%
		7	145	2	19	0.36	3.59	291	n	y	0.97	17.70%
		8	138	2	14	0.87	1.25	-334	n	n	1,510.80	40.12%
		∗ 9	95	4	11	1.11	7.61	-52	y	n	1.00	0.00%
		10	332	3	18	0.99	0.18	-780	y	y	63.75	73.30%
Fixed Size	**AllSol**	11	1198	4	16	1.00	8.35	799	n	n	103.05	7.73%
		12	1049	4	16	1.00	9.74	832	n	n	98.89	5.45%
		13	1148	4	16	1.00	9.33	826	n	n	107.16	6.87%
		14	1057	4	16	1.00	9.00	811	n	n	89.80	13.29%
		15	1005	4	16	1.00	8.36	794	n	y	87.45	11.21%
		16	1115	4	16	1.00	7.20	764	n	y	92.09	8.51%
		17	989	4	16	1.00	7.18	757	y	n	87.79	10.54%
		18	980	4	16	1.00	8.59	808	n	n	102.17	6.20%
		19	954	4	16	1.00	9.94	840	y	y	**109.75**	4.05%
		20	1168	4	16	1.00	7.73	786	n	y	100.21	7.66%
	PerTuple	∗ 11	89	4	16	1.00	5.13	-62	y	n	1.00	0.00%
		12	120	2	16	1.00	1.22	-361	y	y	311.41	35.37%
		13	82	2	16	1.00	2.21	-276	y	y	69.74	28.92%
		14	60	2	16	1.00	3.00	-265	n	n	47.06	15.05%
		‡ 15	33	5	16	1.00	0.46	757	y	n	1.00	0.00%
		† 16	173	3	16	1.00	0.14	-950	y	n	40.36	102.40%
		‡ 17	101	7	16	1.00	9.91	-377	n	n	1.00	0.00%
		‡ 18	131	8	16	1.00	4.41	292	n	n	1.00	0.00%
		19	188	2	16	1.00	0.74	-481	n	n	**1,300.63**	27.01%
		† 20	176	3	16	1.00	0.18	-967	n	n	69.71	116.22%

*: all instances timeout, †: one or two instances crash, ‡: all instances crash

Figures 4.2 and 4.3 show the improvements over the course of the configuration. Configuring for AllSol tends to see smaller improvements being made, while PerTuple makes fewer, large improvements. AllSol configuration also tends to find improvements early on. In all cases, there is significant variation between seeds. However, configuration runs of AllSol with fixed problem size all converge by the end.

The parameter settings obtained by the configuration processes give insight into what problems each algorithm works well on. Two parameters in particular seem correlated with the algorithm speedups: r and δ. When configuring for AllSol, r is set at an average of around eight (8) and δ is a large positive value around 600. For PerTuple, r is generally lower, around three (3), and δ a large negative value around -200. There are exceptions to this, such as adjustable-PerTuple-7 or fixed-PerTuple-15, but those exceptions are almost all poor speedups or crashed runs.

Figure 4.4 shows the effect of both r and δ on the performance of the algorithms. The data shown here includes the intermediate configurations tested on the way to the final configurations. Figure 4.5 shows the combined effect of the parameters.

All of the AllSol configurations of fixed problem size end up with extremely similar parameter configurations as was hinted at by the convergence of their speedups. By restricting the parameters, fewer paths through the parameter space provide viable speedups. Thus, the configuration runs tended to converge.

The r parameter sets the number of constraints while δ influences the number of allowed tuples for constraints. Thus, a configuration with a small r and negative δ yields significantly under-constrained problems, while a configuration with a large value of r and positive δ produces a highly constrained problem. Not only are we able to produce problems favoring both algorithms, we can also determine what makes them ‘favorable.’

4.4 Summary

In this chapter, we have confirmed that pockets of CSPs exist that favor both PerTuple and AllSol. We have also identified the parameters leading to this situation and generated such problems. The parameter settings important to this algorithm choice are the parameters that control the number and tightness of constraints. Highly constrained problems favor AllSol and under-constrained problems favor PerTuple, consistent with our intuitions of the algorithms.

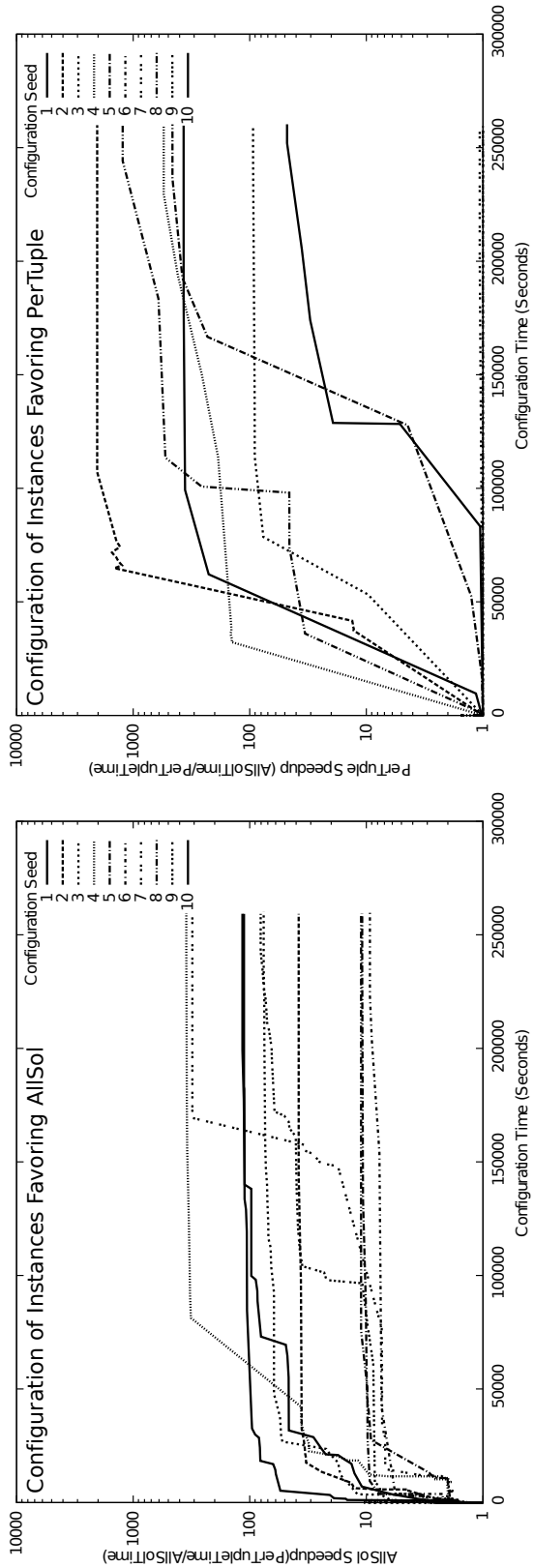

Figure 4.2: Configuration improvement with adjustable problem size.

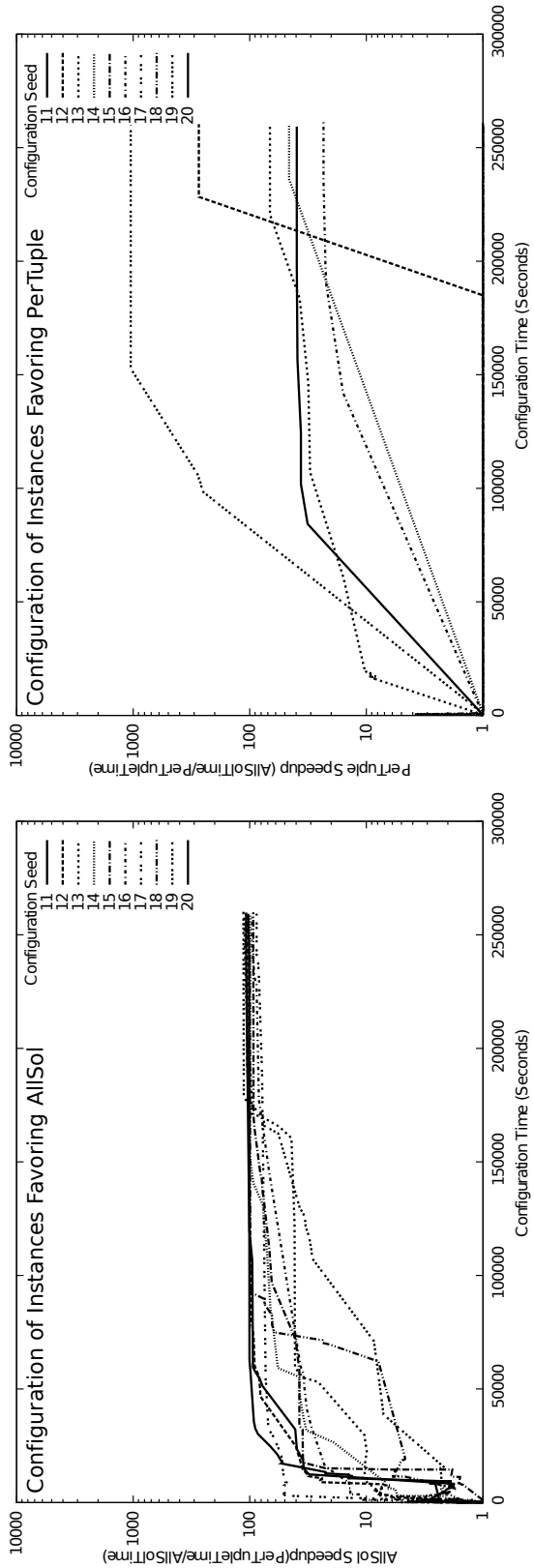

Figure 4.3: Configuration improvement with fixed problem size.

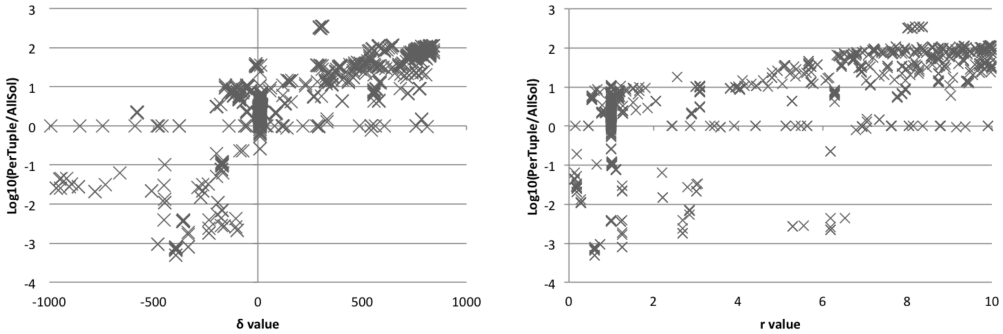

Figure 4.4: Effects of r and δ.

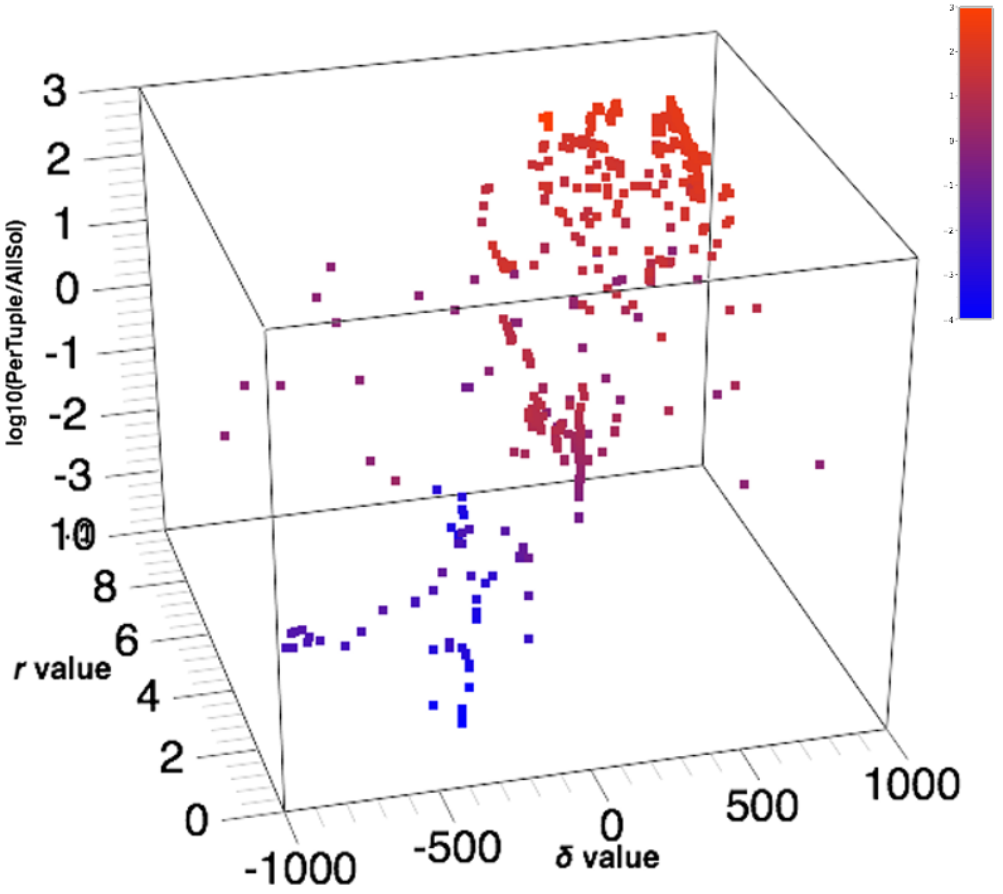

Figure 4.5: Combined effects of r and δ.

Chapter 5

Conclusions and Future Work

The algorithms available to manipulate constraint satisfaction problems are countless and varied. It is not often clear when one algorithm may outperform another. Determining this in an automated manner is a worthy endeavor. In this thesis we have addressed two approaches and applied them to two constraint minimality algorithms: PerTuple and AllSol.

We have shown several machine learning classifiers that classify problems to use either PerTuple or AllSol. We found that the J48 classifier performed the best classification of our data particularly when training with the weighted data set. The weighting of the dataset emphasizes the importance of the most extreme cases and allows them to be properly classified. This classifier saved 99.96% of the time possible to save.

In our investigation of a larger set of benchmarks, we identified an imbalance in the number of problems favoring AllSol and those favoring PerTuple. In the 86 benchmarks examined, PerTuple was much more represented both in the number of instances and the extent to which it was favored. This imbalance could cause a biased classifier and thus led us to search for examples where AllSol performed stronger.

We made use of an algorithm configurator in conjunction with a random CSP generator to build problems better suited to AllSol or PerTuple. This technique proved successful and we were able to obtain problems that ran 100 times faster on AllSol than on PerTuple and problems that ran 1000 times faster with PerTuple than with AllSol. These problems were generated automatically without any human input in the configuration.

After obtaining the problems, we examined the responsible parameters. The parameters r and δ closely correlate to the performance of the two algorithms on the generated problems. These parameters control the number of constraints and the tightness of the constraints respectively. Problems with more constraints and tighter constraints performed better with AllSol while problems with less constraints and looser constraints performed better with PerTuple.

There are several directions for future research:

1. Our technique for random CSP parameter configuration can be extended to tune for many more algorithms. Any two algorithms for enforcing the same consistency property may be dropped into the framework and it will be possible to build problems favoring one over the other. Algorithms for enforcing different consistencies are less straightforward to compare, but still possible. In this case, you may compare the summed runtime of both the propagation algorithm and the subsquent search.

2. Algorithm configuration benefits from having access to many parameters to tune. The RBGenerator exposes several meaningful parameters, but a random generator with more could be useful. For example, it may be useful to have access to parameters that introduce asymmetry in the CSP, control clustering, or add bottlenecks. More parameters provides much more flexibility to the

configurator.

3. Examining the final parameter settings after configuration can give insight about what causes problems to run faster with certain algorithms. We have seen what parameters matter when choosing between AllSol and PerTuple. Next, we can make use of that knowledge in building our classifiers. By looking specifically for the problem features we identify to be important, we can better categorize the problems.

4. In our initial classification experiments, our class imbalance affected the training of the classifier. Now that we have shown how to generate instances favoring AllSol, we can train a classifier on a more balanced set of data. This will allow it to better discriminate between the two classes of problems.

5. Our long-term goal is to develop a large portfolio of propagation algorithms and a robust classifier to select the most useful consistency property to enforce, and for a given property, the most effective propagation algorithm. This selection will occur dynamically during the search in order to allow the system to adapt to the changing needs of the problem. Such a system would allow us to make the best use of the many propagation algorithms available to us.

We have demonstrated feasible approaches for both the selection of propagation algorithms with machine learning classifiers and the tuning of random problem generation with algorithm configuration. Both show promise and will put propagation algorithms to more optimal use.

Bibliography

[Balafrej *et al.*, 2013] Amine Balafrej, Christian Bessiere, Remi Coletta, and El Houssine Bouyakhf. Adaptive Parameterized Consistency. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming*, pages 143–158, 2013.

[Birattari *et al.*, 2002] Mauro Birattari, Thomas Stützle, Luis Paquete, and Klaus Varrentrapp. A Racing Algorithm for Configuring Metaheuristics. In *Proceedings of the Genetic and Evolutionary Computing Conference*, volume 2, pages 11–18, 2002.

[Chmeiss and Sais, 2004] Assef Chmeiss and Lakhdar Sais. Constraint Satisfaction Problems: Backtrack Search Revisited. In *Proceedings of 16th IEEE International Conference on Tools with Artificial Intelligence*, pages 252–257, 2004.

[Elkan, 2001] Charles Elkan. The Foundations of Cost-sensitive Learning. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 973–978, 2001.

[Epstein *et al.*, 2005] Susan Epstein, Richard Wallace, Eugene Freuder, and Xingjian Li. Learning Propagation Policies. In *Proceedings of the Second International Workshop on Constraint Propagation and Implementation*, pages 1–15, 2005.

[Gent *et al.*, 1996] Ian P. Gent, Ewan MacIntyre, Patrick Prosser, and Toby Walsh. The Constrainedness of Search. In *Proceedings of the Thirteenth AAAI Conference on Artificial Intelligence*, pages 246–252, 1996.

[Gottlob, 2012] Georg Gottlob. On Minimal Constraint Networks. *Artificial Intelligence*, 191-192:42–60, 2012.

[Hall *et al.*, 2009] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The WEKA Data Mining Software: An Update. *ACM SIGKDD Explorations Newsletter*, 11(1):10–18, 2009.

[Huberman *et al.*, 1997] Bernardo A. Huberman, Rajan M. Lukose, and Tad Hogg. An Economics Approach to Hard Computational Problems. *Science*, 275(5296):51–54, 1997.

[Hutter *et al.*, 2009] Frank Hutter, Holger H. Hoos, Kevin Leyton-Brown, and Thomas Stützle. ParamILS: an Automatic Algorithm Configuration Framework. *Journal of Artificial Intelligence Research*, 36(1):267–306, 2009.

[Hutter *et al.*, 2011] Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Sequential Model-based Optimization for General Algorithm Configuration. In *Learning and Intelligent Optimization*, pages 507–523. Springer, 2011.

[Karakashian *et al.*, 2012] Shant Karakashian, Robert J. Woodward, Berthe Y. Choueiry, and Stephen D. Scott. Algorithms for the Minimal Network of a CSP and a Classifier for Choosing Between Them. TR-UNL-CSE-2012-0007, 2012.

[Karakashian *et al.*, 2013] Shant Karakashian, Robert J. Woodward, and Berthe Y. Choueiry. Improving the Performance of Consistency Algorithms by Localizing and

Bolstering Propagation in a Tree Decomposition. In *Proceedings of the Twenty-seventh AAAI Conference on Artificial Intelligence*, pages 466–473, 2013.

[Montanari, 1974] Ugo Montanari. Networks of Constraints: Fundamental Properties and Applications to Picture Processing. *Information Sciences*, 7:95–132, 1974.

[Rice, 1976] John R. Rice. The Algorithm Selection Problem. *Advances in Computers*, 15:65–118, 1976.

[Stergiou, 2009] Kostas Stergiou. Heuristics for Dynamically Adapting Propagation in Constraint Satisfaction Problems. *AI Communications*, 22(3):125–141, 2009.

[Woodward *et al.*, 2011] Robert J. Woodward, Shant Karakashian, Berthe Y. Choueiry, and Christian Bessiere. Reformulating the Dual Graphs of CSPs to Improve the Performance of Relational Neighborhood Inverse Consistency. In *Proceedings of the Ninth International Symposium on Abstraction, Reformulation and Approximation*, pages 140–148. AAAI Press, 2011.

[Xu *et al.*, 2007] Ke Xu, Frederic Boussemart, Fred Hemery, and Christophe Lecoutre. Random Constraint Satisfaction: Easy Generation of Hard (Satisfiable) Instances. *Artificial Intelligence*, 171(8):514 – 534, 2007.

[Xu *et al.*, 2008] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. SATzilla: Portfolio-based Algorithm Selection for SAT. *Journal of Artificial Intelligence Research*, 32(1):565–606, 2008.