Metareasoning About Propagators for Constraint Satisfaction

A Thesis Submitted to the
College of Graduate Studies and Research
in Partial Fulfillment of the Requirements
for the degree of Master of Science
in the Department of Computer Science
University of Saskatchewan
Saskatoon

By
Craig D.S. Thompson

Permission to Use

Abstract

Given the breadth of constraint satisfaction problems (CSPs) and the wide variety of CSP solvers, it is often very difficult to determine *a priori* which solving method is best suited to a problem. This work explores the use of machine learning to predict which solving method will be most effective for a given problem. We use four different problem sets to determine the CSP attributes that can be used to determine which solving method should be applied. After choosing an appropriate set of attributes, we determine how well j48 decision trees can predict which solving method to apply. Furthermore, we take a cost sensitive approach such that problem instances where there is a great difference in runtime between algorithms are emphasized. We also attempt to use information gained on one class of problems to inform decisions about a second class of problems. Finally, we show that the additional costs of deciding which method to apply are outweighed by the time savings compared to applying the same solving method to all problem instances.

Acknowledgements

I would like to sincerely thank Dr. Michael Horsch for his supervision, kindness, humour, and support. Mike has been an invaluable instructor and an insightful research advisor. I would also like to thank the Department of Computer Science for their financial support.

Contents

List of Tables

List of Figures

List of Abbreviations

CSP	Constraint Satisfaction Problem
FC	Forward Checking
AC	Arc Consistency
ccs	Constraint Checks
RLFAP	Radio Link Frequency Assignment Problem
QCP	Quasigroup Completion Problem
QWH	Quasigroup with Holes
DWO	Domain Wipe Out
MAC	Maintaining Arc Consistency
FC-CBJ	Forward Checking with Conflict Directed Backjumping
mc	min-conflicts
MGAC	Maintaining Generalized Arc Consistency
EA	Execution Architecture
Multi-TAC	Multi-Tactic Analytic Compiler
KNN	*k*-Nearest Neighbours
LPP	Learning Propagators through Probing
QBF	Quantified Boolean Formula
FORR	For the Right Reasons
ACE	Adaptive Constraint Engine
SCOPE	Search Control Optimization of Planning through Experience
FOIL	First Order Inductive Learner
STRIPS	Stanford Research Institute Problem Solver
ADL	Action Description Language
HAP	Highly Adjustable Planner
ILS	Iterated Local Search
QBF	Quantified Boolean Formula
SIN	Social Insurance Number
MDL	Minimum Description Length
CFS	Correlation-based Feature Selection
MR	Metareasoning

Chapter 1

Introduction

The purpose of this work is to examine the value of metareasoning for solving constraint satisfaction problems (CSPs).

Constraint satisfaction problems can represent a diverse range of problems in many domains including planning, scheduling, circuit design, and software verification. CSPs provide a generalized framework for representing many types of problems, which can all be solved using generalized CSP solving techniques. Solving a constraint satisfaction problem is either done by complete search, or stochastic local search. Complete search involves systematically considering all possible variable assignments until either a solution is found or the set of possible solutions is exhausted. Many techniques exist to either reduce the size of the search tree, or guide the search first in promising directions. Local search by contrast involves repeatedly guessing a complete solution and making iterative adjustments, or repairs, until a solution is found, or a pre-specified number of guesses is attempted. In this work, we focus on complete search methods.

One of the methods used to reduce the search space of a CSP is propagation. Propagation is the process of inferring the effects of the current partial solution on to the remaining variables. In effect, after variable assignments are made, some of the remaining variables may contain values that are not consistent with the current partial solution. Propagation removes these inconsistent values ensuring that effort will not be wasted on attempting to include them in a solution. Two common propagation methods are Forward Checking (FC) and Arc Consistency (AC). Forward Checking ensures that after each variable assignment, all remaining values are consistent with the variable most recently assigned. As a consequence, when Forward Checking is used, after a value is selected from the active domain it does not need to be checked against the other variable assignments in the current partial solution. Arc Consistency, in addition to performing the checks done by FC, will ensure that for each value in each variable, there is at least one value in the domain of each other variable that is consistent. To illustrate, it may be the case that two remaining variables have non-empty domains but their domain values are incompatible. FC will not detect this, whereas AC will. AC does more work after each variable assignment, but can produce smaller search trees. FC does less work after each assignment and thus can explore more tree nodes in the same amount of time. Neither of AC or FC dominates the other; for some types of problem FC tends to solve the problem in less time and for other instances AC is faster (Sabin & Freuder, 1994; Bessière & Régin, 1996).

It would be extremely useful to know *a priori* which algorithm will work best for a given problem and several researchers have studied the task of algorithm selection (Epstein, Wallace, Freuder, & Li, 2005; Pulina & Tacchella, 2009; L. Xu, Hutter, Hoos, & Leyton-Brown, 2008). Choosing an appropriate solving method for a CSP is important as the difference in runtime between methods can be several orders of magnitude. Poor choices when choosing a solving method can result in wasted resources by applying an inefficient solver. Furthermore, several specialized solvers have been developed to take advantage of the structure of particular problems or classes of problems. However, designing a problem-specific solver becomes increasingly difficult as increasingly complex problems are considered. The active research in algorithm selection and special case algorithm design indicates that there is value in being able to know when to apply which solver. To address this problem, we use machine learning to predict which CSP solving method will be most efficient for solving a given CSP. Our work focuses on choosing the best algorithm on a per problem basis (rather than configuring an algorithm to suit a class of problems) and determining which problem features are indicative of the solving method that should be used.

Given a set of CSP solvers, an oracle will always apply the optimal solver for a given problem. Furthermore, we expect that there exists a solver in the set whose average performance over a range of problems is better than the average performance of any other solver in the set. We know that there are specific problem types that favour one solver over another. Therefore, there is no single solver that will dominate over the entire range of CSP problems and we can conclude that an oracle solver is preferred to the solver with best average performance. In hindsight, we can see which solver is most effective for each problem instance. However, we aim to improve the foresight of predicting which algorithm will be most effective by using machine learning. When attempting to predict which solver will be most effective we aim to approximate an oracle, and performance equal to the best average solver can be considered a baseline for usability.

One of the problems we face is in determining how CSPs are alike or different. One solution to this problem often explored in the literature is to assume that a group of like problems needs to be solved (Epstein et al., 2005; Minton, 1996). Thus, the task is to configure or select an algorithm believed to be effective on this group. In essence, this avoids the task of classification, as the problems are assumed to be similar. However, we are interested in selecting an algorithm on a problem by problem basis. Therefore, we must have a method for determining if a newly encountered problem is similar to problems we have previously encountered. CSPs can be alike in many ways, such as the number of variables contained in the problem instance, or the number and type of constraints. However, the measures of a CSP that we are interested in are the ones which can be used to differentiate CSPs based on the preferred solving methods. There are many possible

CSP solving algorithms, so in our work we simplify the problem by considering only two solving variants, AC and FC. We aim to group CSPs into those that are solved faster with Arc Consistency and those that are solved faster with Forward Checking. Thus, when a new problem is encountered we must determine if it is like those that are solved faster with one algorithm or the other.

There are several benefits of our approach: first, we demonstrate the ability to learn to select an appropriate solver based on quickly measurable problem features; second, the overhead of selecting an algorithm is small relative to the time required to solve the CSP, and performance improvements are significant; third, the method works effectively on heterogeneous problem groups. Our work provides some insight into the problem of algorithm selection for CSPs, but there is still significant work to be done in considering additional solver configurations and problem types.

The rest of this thesis is structured as follows: in Chapter 2 we survey the literature on constraint satisfaction techniques, including propagation methods as well as variable and value selection heuristics, and the recent literature on metareasoning, particularly in the domain of combinatorial problems; Chapter 3 contains our methodology used for solving CSPs, the learning algorithms we used and a description of our datasets; in Chapter 4 we present our experiments and results; finally, in Chapter 5 we present our concluding remarks and directions for future research.

Chapter 2

Literature Review

In this chapter, we will discuss both classical and modern approaches to solving CSPs, as well several approaches to metareasoning.

2.1 Constraint Satisfaction

In this section, we will review constraint satisfaction concepts including: basic definitions, example CSPs, attributes of constraint satisfaction problems, as well as basic and advanced solving techniques. As we will see, there are many types of CSPs and many solution methods. However, in our experimentation we use three types of constraint satisfaction problems: random, small world graphs with random constraints, and quasigroups with holes, each of which is described in this section. Furthermore, we will survey many solving techniques, but our experimentation involves two in particular: Forward Checking and Arc Consistency.

2.1.1 Definitions

A constraint satisfaction problem P is defined as a triple $P = (X, D, C)$, where X is a finite set of variables $x_1, \ldots, x_n$, D is a set of finite domains $d_1, \ldots, d_n$, one for each variable in X, and C is a finite set of constraints. Each constraint consists of a set of variables, $var(c)$, and a relation, $rel(c)$, consisting of a subset of the Cartesian product of the domains of the variables in $var(c)$. As an example, a constraint c_1 between variables x_i and x_j has $var(c_1) = \{x_i, x_j\}$ and $rel(c_1)$ is of the form $rel(c_1) \subseteq d_i \times d_j$. The constraints are used to limit the combinations of variables that are allowed in solutions to the problem. The process of checking if a given value tuple is allowed by a constraint is called a *constraint check*. For simplicity, we only consider constraints between pairs of variables. Constraints between more than two variables can be transformed to binary constraints over a new set of variables either by dual encoding or hidden variable encoding (Bartak, 2001). CSPs can be represented as graphs, where the variables correspond to graph nodes and the constraints correspond to graph edges. We refer to the neighbourhood of a variable x_i as $\Gamma(x_i)$, the set of nodes sharing an edge with node x_i. As an example, the map colouring problem is to assign a colour to all the regions of a map such that no regions sharing a boundary are assigned the same

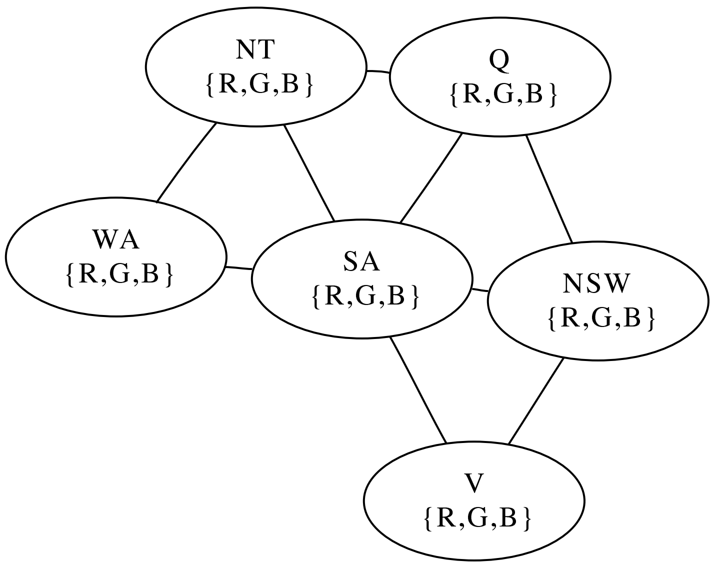


Figure 2.1: Constraint graph of the Australia map colouring problem. Variables correspond to the states and territories of Australia. Each variable must be assigned a colour from the set {Red, Blue, Green} such that no adjacent regions are assigned the same colour

colour. In this problem, the variables correspond to the regions of the map, the domain values are the set of allowable colours, and there are constraints imposed between all adjacent regions such that their colours are not equal. In Figure 2.1, we present the constraint graph for colouring the states and territories of Australia, adapted from (Russell & Norvig, 2003).

A partial solution to a CSP is an assignment to some variables such that no constraint in C is violated. It may or may not be possible to extend each partial solution to a solution. A solution to a constraint satisfaction problem binds each variable in X to a value in the domains D such that no constraint in C is violated. The constraint satisfaction problem is to find a value in the domain of each variable such that all constraints are satisfied. There may be many, one, or zero satisfying assignments, and the number of satisfying assignments is typically not known *a priori*.

At the most general level, there are two types of CSP solving algorithms, complete and incomplete. Complete algorithms explore the search space systematically, guaranteeing that if a solution exists it will be found, and if a solution does not exist all possible solutions will be considered. Incomplete algorithms typically involve a randomized exploration of the search space whereby a full assignment is made to all variables, the number of violated constraints is counted, and the assignment is iteratively refined in an attempt to find a solution. The iterative refinement process is usually limited to a fixed number of adjustments. After which, if a solution has not been found a new full assignment will be made, and the process is repeated for some fixed number of trials. Incomplete algorithms have been shown to perform very well on problems that have solutions, but cannot be used to prove that a problem has no solution. We have chosen to focus on complete search techniques in our work.

When empirically evaluating CSP solvers there are several metrics commonly cited in the literature. The most general measure of performance is the time taken to solve CSP. Solving time is a useful measure because in most applications speed is the primary concern (after correctness). However, solving time is hardware dependent, and as such makes comparing new results to previously published work more difficult. Furthermore, solving time does not provide any insights into how an algorithm explores the search space. Three more metrics are often cited: the number of constraint checks performed (ccs), the number of search tree nodes visited (nodes), and the number of time the search algorithm has to backtrack (backtracks). These metrics are useful for comparing new algorithms to old, as an old algorithm running on new hardware will still produce the same results as it did previously. While ccs, nodes, and backtracks provide insight into how an algorithm explores the search space, they do not give an accurate indication of "real world" performance, in the same way that solving time does. Furthermore, many of the specialized variants of Arc Consistency aim to speed up the process of checking constraints by storing additional information. As such, two algorithms with the same number of constraint checks, nodes visited and number of backtracks may not take the same amount of time. As we are interested in the process of choosing an algorithm to improve performance, we measure using time, rather than ccs, nodes, or backtracks.

2.1.2 Example Problems

In this section, we will describe some of the problems classes cited in the literature, as well as those used in our experimentation. Broadly, CSP examples can be divided into those that are "found", often by encoding some "real world" problem, and those that can be generated according to some set of parameters. Several repositories of CSPs exist including C. Lecoutre's website (`www.cril.univ-artois.fr/~lecoutre/`), `www.csplib.org`, and the DIMACS problem set at `dimacs.rutgers.edu/Challenges`. As previously mentioned, our experimentation focuses on random problems, small world graphs with random constraints imposed on each edge, and quasigroups with holes; many other types of problems are described in this section for review.

Generated problem types are of interest because they can be tuned in size and shape to fit the requirements of experiments. If for example, we are interested in studying solver algorithm performance on a particular class of problems, we can generate an arbitrary number of instances meeting the requirements. Furthermore, generated problems can be randomly or pseudo-randomly generated, vastly increasing the number of possible instances. However, it must be noted that performance on random problems should not be considered indicative of "real world" performance, rather they are only used as a convenience for experimentation. Constraint satisfaction problem hardness is determined by many factors including the number of variables, domain size, constraint density, and constraint tightness. Additionally, the arrangement of the constraints among the variables plays an important role in problem hardness.

Random CSPs are defined by the number of variables n, the domain size of each variable m, the number of randomly generated constraints, and the tightness (defined below) of the constraints. The domain size is typically uniform for each variable. The constraints may either be added exactly, to guarantee the exact number of constraints in the problem, or probabilistically, adding a constraint between every pair of variables in accordance with a given probability. A control parameter, p_1, denotes the constraint density, or the number of constraints in the problem as a fraction of the maximum possible number of constraints in the problem. Constraint density is a measure of how connected the constraint graph is. Using an exact method to generate the constraints, rather than a probabilistic method will result in a more uniform distribution of problems generated from the same parameters. Constraint tightness, controlled by a parameter p_2, denotes the fraction of domain value tuples that are disallowed by each constraint. Constraint tightness can also be determined exactly, or probabilistically. Problems with constraints generated probabilistically may exhibit local tight and loose regions, whereas exact generation will produce uniformly tight constraints. Random CSPs are typically defined by their $\langle n, m, p_1, p_2 \rangle$ values. Four classes of random problems are defined by Gent et al. and presented in Table 2.1 depending on whether the constraint density and tightness are generated exactly or probabilistically (Gent, MacIntyre, Prosser, Smith, & Walsh, 2001). Problems generated with low p_1 and low p_2 almost always have a solution, whereas problems with high p_1 and high p_2 almost always do not have solutions. The area of the problem space in between these extremes, called the "mushy" region contains some soluble and some insoluble problems (Smith, 1994).

Table 2.1: Random Problem Classes

Problem Class	p_1	p_2
Model A	Probabilistic	Probabilistic
Model B	Exact	Exact
Model C	Probabilistic	Exact
Model D	Exact	Probabilistic

Geometric problems are a variant of random problems where all variables are randomly assigned a point within a unit square and constraints are added between all pairs of variables where the distance between the variables is less than some value d. Whereas Random CSPs are truly random in their structure, geometric problems will exhibit local structures without "short cuts" across the CSP.

Model RB is a variant of model B used by K. Xu and Li (2000). Model B consists of randomly selecting exactly the fraction p_1 of the possible edges and picking exactly p_2 of the pairs as no goods. Model RB adds two parameters α and r to control the number of constraints and domain size relative to n.

Gent et al. (2001) introduce “flawless” constraints and demonstrate their use. The authors define a flawed value as one that violates a constraint with a neighbouring variable. A variable is flawed if all of its values are flawed, and thus there is no possible assignment to that variable that will produce a solution. The authors present evidence that as the number of variables n increases, so to do the odds of there being a flawed variable, if $p_2 \geq 1/m$. If $p_2 \leq 1/m$, then problems will not be made trivially insolvable as other problem parameters increase. For the parametrization $\langle 100, 3, p_1, 4/9 \rangle$ the authors find a much broader “mushy” region when using flawed constraints (a larger region with some solvable and some unsolvable problems) and problems begin to be unsolvable with a much lower p_1. They also find that peak hardness (measured as the median number of constraint checks) is much higher for flawless constraints and hardness remains higher for over constrained problems. Gent et al. present a method for producing “flawless” problem instances by ensuring each value in the domain is supported by at least one value in the domain of the second variable. As a consequence, AC preprocessing of “flawless” problems produces no domain reductions. The authors also present a comparison of problem hardness using flawed and flawless constraints in a region of the problem space with very low probability for flawed variables. They find that there is virtually no change in the phase transition location, or peak hardness, indicating that flawed and flawless constraints produce problems of equal difficulty when there are no flawed variables.

Gent et al. (2001) also present results considering a structured graph, with random constraints. For example, using quasigroup completion problems, and instead of the standard constraints indicating that each colour must be unique in each row and column, adding random constraints. The quasigroup completion problem contains many large cliques, that would be extremely rare in randomly generated problems. They note that these part-real part-random problems help to fill the gap between plentiful random problems and hard to collect real problems; however, it may be the case that introducing random constraints loses some (or all) of character of a real problem, but this remains an open research problem. We make use of this technique to generate small world graphs (Watts & Strogatz, 1998) with random constraints on each edge. We describe the features and generation method for small world CSPs in Section 3.5.3.

The Graph Colouring Problem consists of taking a map of regions and assigning a colour to each region such that no adjacent regions have the same colour. Graph colouring problems may be real, such as colouring a map of the earth, or randomly generated.

Radio link frequency assignment problems (RLFAP) (Cabon, De Givry, Lobjois, Schiex, & Warners, 1999) originate from a French government research problem of setting up radio towers to create a communications network. Each tower needs to be able to communicate with the other towers it is near, and each communication channel between two towers must be assigned a communication frequency. The problem is to assign a frequency to each link such that there is not interference between neighbouring links. Many variations on the RLFAP problem exist including subsets of the original graph and limitations on the number of unique frequencies that may be employed. RLFAP is also a commonly used constraint optimization problem, where channel interference is not strictly forbidden, but must be minimized.

Langford's number problem, L(X,Y), consists of arranging X sets of Y numbers in a sequence such that all occurrences of number y are separated by y spaces. For example, the problem L(2,3) consists of arranging 2 of each of the numbers 1,2,3, such that there are two 1s, two 2s, and two 3s, and the 1s are separated by 1 digit, the twos are separated by two digits, and the threes are separated by three digits. It has a solution: 312132.

Boolean problems are those where each variable may only take on one of two values. The most notable of these is 3SAT. 3SAT problems consist of conjunctions of disjunctions, where each disjunction is a triple of variables. Thus, in its standard form, each constraint exists between the three variables in the disjunction and there are as many constraints as there are conjunctions. Often, 3SAT problems are transformed into binary constraints (where each constraint exists between a pair of variables) before being solved as a CSP. Furthermore, Boolean problems can be generated with arbitrary arity.

A Latin square is an $n \times n$ matrix filled with n distinct symbols such that each symbol appears in each row and column exactly once. The Latin square problem is to determine if a partially completed matrix can be completed to form a Latin square; this is also known as the quasigroup completion problem (QCP). QCPs may have zero, one, or many solutions. Latin squares have applications in experimental design, cryptography, and error correcting codes.

The quasigroup with holes (QWH) problem is different in that we start with a completed Latin square and "poke holes" in it to produce a partial solution (Achlioptas, Gomes, Kautz, & Selman, 2000). The goal is then to fill in the holes to complete the Latin square. QWH problems differ from QCP problems in that QWH problems are guaranteed to have a solution, as the problem was generated from a complete Latin square. For QCP and QWH problems, difficulty increases as the size of the problem and the distribution of initially empty cells, or holes, becomes more balanced. A distribution is called balanced if each row and column has the same number of holes. QWH problems are particularly useful for measuring the quality of an incomplete solver. Previously, generating hard random solvable instances involved generating a problem and then solving the problem with a complete solving method and discarding all unsatisfiable instances. However, local search techniques are known to be effective on problems much larger than those that can be solved by complete methods, so there is a demand for problems beyond the scope of those that can be solved completely.

Solving methods for QCP and QWH problems have been studied by Gomes and Shmoys (2002).

The authors compare three solving techniques (SAT, CSP, and hybrid linear programming/CSP) and found that none of the three techniques was superior for all problems types, with the fastest solver depending on problem size and how constrained the problem was. This finding indicates that proper problem identification and solving algorithm selection could speed the solving process for QCP and QWH.

The n-queens problem is based on chess, where the goal is to place n queens on an $n \times n$ chess board such that no queen is attacking any other queen. Satisfying placements exist for $n = 1$ and $n \geq 4$.

The k-knights problem is to place k knights on a $k \times k$ chess board such that the position of the knights form a cycle (when considering knight moves) (Boussemart, Hemery, Lecoutre, & Sais, 2004). This problem has no solution for any odd number of knights.

2.1.3 Special Case CSPs

There are a few CSP graph structures that are worth pointing out as there are particular methods suited to them. The problem structure of a CSP plays a major role in the time complexity of finding a solution. First, if there are any independent components in the problem (that is, if the constraint graph consists of multiple disconnected components) then each part can be solved independently. For example, if a problem can be split in half, the size of the search space can be reduced from d^n to $2d^{n/2}$, where d is the domain size and n is the number of variables.

Second, constraint graphs that take the form of trees (that is, graphs that are acyclic) can be solved in polynomial time. The algorithm is as follows (Freuder, 1982): choose any node in the tree to be the root of the search space and order the nodes such that the parent of a node in the constraint tree precedes it in the search tree; apply Arc Consistency starting from the last variable in the ordering going to the first node; finally, a non-backtracking search can be applied from the root node down the chosen variable ordering. The efficiency of constraint trees comes from only needing to apply consistency to each constraint once, as each child node is only constrained by its parent. Furthermore, once Arc Consistency is applied, any paths remaining in the search tree will produce a solution. This algorithm runs in $O(nd^2)$ time, which is a large improvement compared to the possible size of the search space, $O(d^n)$.

Given that there is an efficient algorithm for solving tree shaped CSPs, reducing other CSPs to trees can also be effective. One such method involves finding the *cutset* of a graph, the set of nodes that once removed, reduce the graph to a tree. We can then find all solutions to the CSP problem for the variables in the cutset, and for each solution to the cutset, remove the inconsistent values from the remaining nodes and use the tree algorithm above. If any of the possible cutset solutions lead to a solution to the full problem, we have solved the entire problem. The complexity of this solution method, $O(d^c(n-c)d^2)$, depends on the size of the cutset found, where d is the domain size, n is the number of variables and c is the size of the cutset (Russell & Norvig, 2003).

2.1.4 Backtracking Search Techniques

The search space for a constraint satisfaction problem is bounded from above by d^n, where d is the number of values in a variables domain and n is the number of variables. If the variables in n have dissimilar domain size, the product of their domain sizes is the bound on the search space. Because a solution to a CSP is not dependent on the order in which variables are considered we can choose an ordering which will minimize the size of the search tree. CSPs are typically solved using some form of depth first search, although there are many variants to consider. The most basic form of search used is backtracking search, in which each variable is assigned a value from its domain, one at a time. Each variable is assigned a value that satisfies the constraints between it and the previously assigned variables, if possible. If a satisfying assignment is not possible, the algorithm backtracks to the most recent choice point. In this case the most recent choice point is the assignment of the most recently assigned variable. Once all choices are exhausted at the most recent choice point the algorithm backtracks to the second most recent choice point and continues the search.

Backmarking is a technique used to reduce the number of times the same constraints are rechecked. We maintain a table M for the domain of each variable indicating the earliest variable inconsistent with the given domain value. We also maintain a list *low* indicating the earliest variable we have backtracked to since each variable was last considered. For a given variable x_i, if the value in *low*, low_i, is less than or equal to the variable's value in M, M_i, then all consistency checks between the variable and those above low_i are known to succeed and do not need to be repeated. On the other hand, if low_i is greater than M, then we have not yet backtracked to the variable causing the constraint violation, so no checks need to be done as we know they will not succeed.

Backjumping is another technique for reducing the search effort. As an example, suppose all values in the domain of variable x_i conflict with variables x_1 and x_2, backtracking search will attempt to backtrack to variable x_{i-1}. Backtracking to variable x_{i-1} will result in the same problem when we determine that once again all values for the variable x_i conflict with the first two variables. Backjumping search provides a solution to this problem. When we consider the conflicts in a possible assignment we produce a conflict set, the set of variables which cause conflicts in the domain of the current variable. When the domain of the current variable is exhausted, we backtrack to the most recent variable in the conflict set. In the previous example, we would build the conflict set $\{x_1, x_2\}$ for variable x_i and once the domain was exhausted the search would backtrack to x_2. By skipping back to the most recent conflict we can avoid doing needless work reconsidering values for $x_3 ... x_{i-1}$.

Conflict-directed backjumping is a further refinement of backjumping that can sometimes jump back higher in the search tree, resulting in pruning a greater portion of the search tree. Conflict directed backjumping works by considering whether a constraint between two variables actually caused an inconsistency to be found. The jumpback set of a variable is determined by considering each value in the domain and determining the earliest constraint that forbids the value. Constraint A is earlier than B if the latest variable in $\mathit{scope}(A) - \mathit{scope}(B)$ precedes the latest variable in $\mathit{scope}(B) - \mathit{scope}(A)$, where the scope of a constraint is the set of variables it constrains (Dechter, 2003). Thus, the node one should backjump to is the latest variable in the backjump set. Suppose variable x_i shares constraints with variables x_1, x_2, x_3 and that the assignments made for variables x_1 and x_2 conflict with all values in d_i. Backjumping would jump to x_3 because it is the most recent variable in the conflict set. Conflict-directed backjumping would jump back to x_2, the latest variable in the conflict-directed jumpback set.

Another variation to improve the performance of search is to consider learning new constraints implicit in the problem. Whenever a dead-end is found, we can create a new constraint forbidding the current combination of variable assignments. However, recording the full set of variable assignments is not useful as we will never attempt the same assignment, assuming the search algorithm is properly coded. But, if some subset of the current variable assignments leads to a state where no assignments can be made at the current variable then we can record this subset as a new constraint, as we will likely encounter it again in the search tree (Dechter, 1986).

Backmarking, backjumping, conflict-directed backjumping, and constraint learning all involve a time/space trade-off: we are increasing the storage size by storing extra information about constraints, but we are reducing many constraint checks to simple table look-ups. Balancing the costs and benefits of consistency and search has been a topic of much research. Backtracking and its variants are often called look-back techniques, as they are used to "look back" in the variable assignments to detect where a failure occurred. Next we will consider propagation techniques, which are often called look-ahead techniques, as they are used to "look ahead" in a problem to detect failures before they occur. Both the look-ahead and look-back techniques solve the same problems of avoiding variable combinations that will not lead to solutions. In fact, some of the techniques are exactly equivalent: backjumping will jump back to the most recent variable in the conflict set, and from this variable using Forward Checking we could detect that an dead-end would be found (Russell & Norvig, 2003).

2.1.5 Propagation Techniques

Consistency techniques (also called inference or propagation) are used to reduce the search space in order to reduce the time taken to find a solution. Consistency refers to whether or not a partial assignment satisfies the given constraints. The application of consistency techniques prior to search in a CSP problem can greatly reduce the size of the search tree.

A binary constraint over variables x_i and x_j is said to be *arc consistent* if every $a \in d_i$ is *supported* in d_j. A value a is supported in d_j if there exists a value $b \in d_j$ such that the tuple $\langle (x_i, a), (x_j, b) \rangle$ is allowed by the constraint (Mackworth, 1977a).

One of the most common CSP solving techniques is to interleave inference and search. Typically, this involves choosing a variable to instantiate and choosing a value for that variable and then propagating the effects of that choice throughout the domains of the other variables. A *domain wipe out* (DWO) occurs whenever there are no remaining possible values to assign to a variable. As long as a DWO does not occur, the search process continues. When a DWO does occur, the search algorithm backtracks. The advantage of interleaving inference with search is that inference may cause DWOs to occur earlier, resulting in a smaller search space.

Node consistency is applied to each variable, pruning values that are inconsistent (or, incompatible) with the unary constraints. Node consistency is the easiest and fastest type of consistency to apply. If there are unary constraints in the problem, ensuring node consistency can reduce the search space, especially when used with more complex forms of consistency. Often there are no unary constraints in CSP problems, and instead, the domain sizes are reduced in the problem formulation to accomplish the same task.

If no propagation is interleaved with search, then each time an assignment is made we must check that the most recent assignment is consistent with the current partial solution. Forward Checking is the most limited form of constraint propagation required to ensure we do not need to check if the current assignment is consistent with the previous assignments (Haralick & Elliott, 1980). Forward Checking ensures that all remaining values in the domains of unassigned variables are consistent with the current partial solution. To achieve this, each time a variable is assigned, all constraints between the newly assigned variable and unassigned variables are made arc consistent by deleting values from the domains of the unassigned values that are not consistent with the value of the newly assigned variable. This ensures that all remaining domain values for all variables are consistent with the current partial assignment. Forward Checking ensures that an assignment for the current variable is supported by at least one value in each of the constraints between it and the remaining variables. If there is no support, then there will be a DWO and backtracking will occur. Forward Checking considers the effects of choosing a value for the current variable on each of the remaining variables independently. As such, Forward Checking will not always discover situations where a partial solution cannot be extended to a complete solution. For example, if there is an assignment to the current variable x_i such that there is only one consistent value in the domain of variable x_j and the proposed value for x_j conflicts with all possible values for x_k. In this situation, Forward Checking will detect a DWO and backtrack when assigning x_j, but not when considering x_i. However, Arc Consistency, which we discuss next, will detect that this partial solution cannot be successfully extended when considering x_i. Forward Checking provides some degree of look ahead and is quick because at worst we consider each constraint once per application of FC.

Arc Consistency, when interleaved with search (as is done in Maintaining Arc Consistency (MAC)), can detect that a partial solution cannot be extended earlier than Forward Checking by checking consistency for all unbound variables, rather than just those that share a constraint with the current variable (Sabin & Freuder, 1994). When checking the consistency of a value for x_i, if we find a consistent value in x_j, we must ensure that the value in x_j is supported in the other constraints involving x_j. Additionally, the process is repeated each time a value is removed from one of the domains, as that value may have been used to support one of the previously checked values. Arc Consistency can produce smaller search trees than Forward Checking, but comes at a cost as the number of constraint checks are greatly increased. Furthermore, Arc Consistency also fails to detect some partial solutions that cannot be extended to complete solutions. Using the previous example, the values in the domain of x_k do not have to be consistent with x_i and x_j at the same time. For example, there may be a constraint that each of the variables take a unique value from each other and all variables may have domains *red* and *blue*. In this case, for all of the three variables the domain values are all supported, but not at the same time. To solve this problem we need to consider higher degrees of consistency. Many variants of Arc Consistency exist, each aiming to reduce the upper bound on the number of checks performed, but at a cost of increased space complexity. For example, AC-3 has time complexity $O(ed^3)$ and space complexity $O(ed^2)$, whereas AC-4 has time complexity $O(ed^2)$ and space complexity $O(ed^2)$, where e is the number of constraints and d is the domain size. Although the algorithms have the same asymptotic space complexity, AC-4 requires more space by a constant factor of 2. Arc Consistency is incomplete in that making a CSP arc consistent does not necessarily yield a solution (Tsang, 1993). Additionally, there are Arc Consistency algorithms for non-binary constraints, such as Generalized Arc Consistency (Mackworth, 1977b).

To illustrate the differences in domain reductions when using AC and FC, we present the remaining domain values after two variable assignments to the Australia map colouring problem. In Figures 2.2 and 2.3, we present the CSP after assigning the value R to the variable WA and performing FC and AC respectively. As we can see in this case, there is no difference in the domain reductions, although the Arc Consistency algorithm has done several more constraint checks. However, in Figures 2.4 and 2.5, we assign a second variable (SA=G) and again perform FC and AC respectively. Here, after the second variable assignment we can see that the additional constraint checks done by AC result in fewer remaining domain values than remain when using FC.

There are more complete consistency concepts, such as *path consistency*, that consume more time than Arc Consistency but prune more values. Path consistency ensures that for all groups of three variables x_i, x_j, x_k all consistent pairs in two variables x_i, x_j have a supporting value in the third variable x_k.

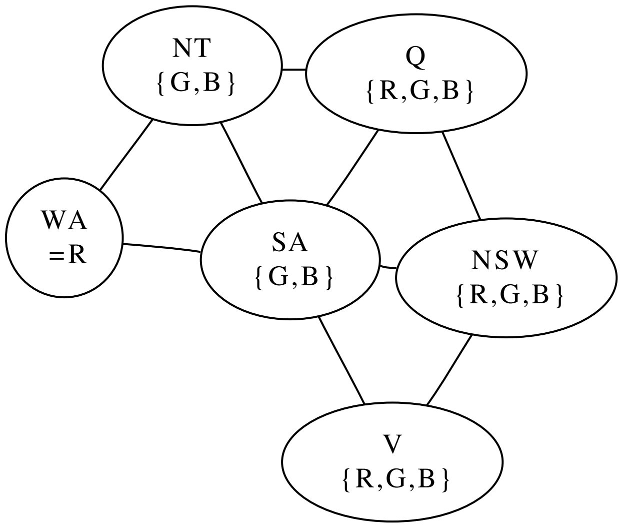


Figure 2.2: Constraint graph of the Australia map colouring problem after the assignment WA=R and performing Forward Checking.

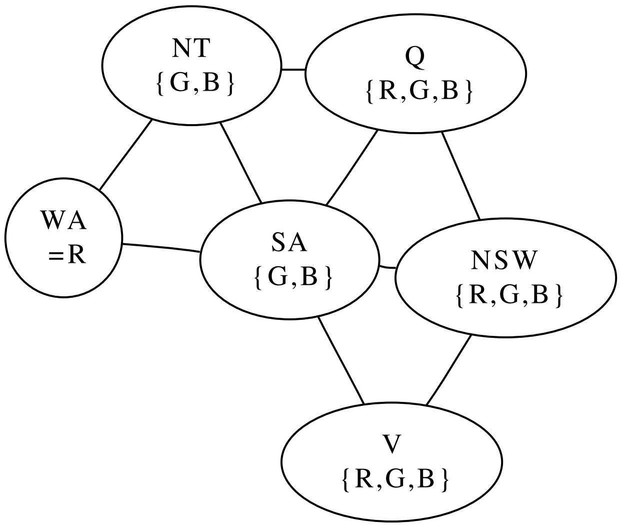


Figure 2.3: Constraint graph of the Australia map colouring problem after the assignment WA=R and performing Arc Consistency.

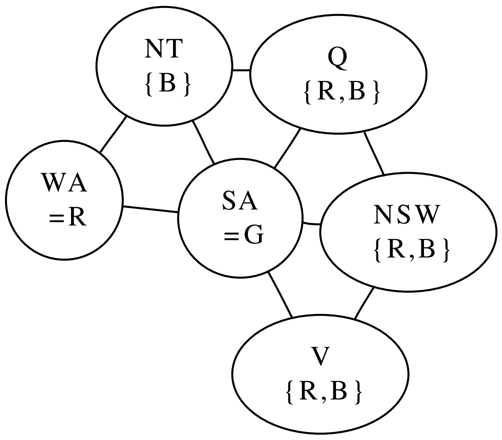


Figure 2.4: Constraint graph of the Australia map colouring problem after the assignments WA=R and SA=G, and performing Forward Checking.

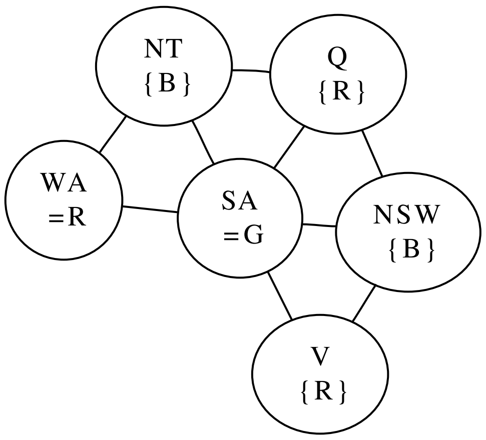


Figure 2.5: Constraint graph of the Australia map colouring problem after the assignments WA=R and SA=G, and performing Arc Consistency.

Node, arc, and path consistency are all generalized by a concept called k-consistency. Node consistency which acts on single variables corresponds to 1-consistency. Arc Consistency or 2-consistency ensures pairs are consistent, and path consistency corresponds to 3-consistency. Higher levels of consistency, such as k-consistency, where $k > 3$ are not typically performed in practice because of the high computational cost. However, there does not seem to be an optimal amount of consistency to apply to problems generally. Rather, it is problem specific, and there is a trade-off between time spent enforcing consistency and time spent considering possible solutions through search.

2.1.5.1 Choosing Between AC and FC

Many researchers have investigated the application of Arc Consistency and Forward Checking in an attempt to determine which method is superior. In this section, we will discuss some of these papers in detail.

Sabin and Freuder (1994) make two contributions: first, by providing evidence that Arc Consistency preprocessing can actually increase the search efforts of Forward Checking, which they call AC+FC. Not only does the combined effort of AC+FC exceed just using FC, but the FC portion of AC+FC takes longer than just using FC. In other words, AC preprocessing can actually make a problem harder to solve, rather than easier. This is in contrast to the previous wisdom that enforcing greater levels of consistency made the remaining problem easier to solve. However, the authors used the minimum domain size (*dom*) variable ordering heuristic (refer to Section 2.1.6 for a discussion of variable ordering heuristics) and state that, when they used a lexicographic variable ordering, AC preprocessing no longer degraded the performance of FC search. This provides confirming evidence to the theory that there can be significant interplay between solver components, and it seems to be the combination of AC preprocessing and *dom* variable ordering that results in the decrease in solving speed.

Sabin and Freuder's second contribution is Maintaining Arc Consistency (MAC), which has since become one of the most widely used CSP solving techniques. MAC involves making a problem initially arc consistent and enforcing Full Arc Consistency after each variable assignment. Additionally, whenever a variable assignment is attempted, if AC finds the problem to be inconsistent, then the value is removed from the domain and AC is enforced before the next domain value is attempted. Thus, each node in the search tree has 2 child nodes: one corresponding to the attempted assignment and one corresponding to the negated assignment (called 2-way branching). Conversely, when not using MAC, the typical branching structure is to have one child node for each possible domain value (called d-way branching). For problems in the "hard" region, MAC is shown to significantly out perform FC by as much as an order of magnitude. The authors note that MAC is especially advantageous over FC for hard problems with low p_1. The authors mention that FC outperforms MAC on easy problems, but from their presented data it is unclear how large this advantage is.

Bessière and Régin (1996) provide further evidence to support the use of MAC, as well as presenting new variable ordering heuristics. Their work fills some of the gaps left by (Sabin & Freuder, 1994) by using model B random problems instead of model A. They use these changes to reduce the variance with problems with the same parameters in the hopes of reducing the number of problems needed to be solved to get a representative understanding of performance. The authors chose to use the same problem parameters as Frost and Dechter (1995) to compare MAC and FC-CBJ (Forward Checking with conflict-directed backjumping), both using *dom+deg* variable ordering (*dom*, breaking ties with *deg*) and min-conflicts (*mc*) value ordering (refer to section 2.1.7 for a discussion of value ordering). The authors show that FC does 1.5× to 230× more constraint checks, takes 2× to 476× longer, and does 7× to 870× more backtracks than MAC on the selected problems. The authors systematically explore the random problem space and show that MAC outperforms FC by a wider margin under three circumstances: as the problem approaches peak hardness, as domain size increases, and as constraint density decreases.

2.1.5.2 Refinements to Propagation Techniques

In this section, we will explore some of the modern variants of Arc Consistency. Generally, improvements to Arc Consistency consist of storing some additional information in order to reduce the number of constraint checks performed.

Bessière, Régin, Yap, and Zhang noted that, when enforcing Arc Consistency, a considerable number of constraint checks could be avoided by storing the support values found and when beginning a new support check, checking if the old supports are still available (2005). If the previous support is still available, then it can be used. Otherwise, we can look for support in the domain values *after* that value, as we have previously examined all values earlier in the domain than the last support and none were found suitable. This variant of Arc Consistency is named AC2001/3.1. A later work by Likitvivatanavong, Zhang, Shannon, Bowen, and Freuder implemented AC2001/3.1 as well as three similar variants, confirming that storing previous supports can save time during search (2007).

Two more methods of storing previous supports are presented by Lecoutre and Hemery (2007), AC3r and AC3rm. AC3rm is *multi-directional* in that whenever a support y is found for a value x, we can store the support in both directions (i.e y supports x and x supports y). Integrating AC3rm with MAC is simpler than AC2001/3.1, as backtracking does not induce any complex unwinding of the data structure that stores supports; a support at a deeper level of the search tree may still be a support when we backtrack and try a different branch (this was shown using AC3.2 by Lecoutre, Boussemart, and Hemery (2003)).

An alternate method of improving AC performance is to control the order in which constraints are revised, as is done by Balafoutis and Stergiou (2008b). AC maintains a queue of constraints to be revised called the *revision list*. Previous literature explored the use of *dom* or *deg* heuristics to guide the ordering of the revision list, but this work is novel in that it explores the use of weighted constraints and related metrics such as *dom/wdeg*. The authors propose several new revision list ordering heuristics, the most effective of which is *dom/wdeg* (choose the arc in the revision list involving the variable with the smallest ratio of *dom/wdeg*). There is very little overhead in using *wdeg* or *dom/wdeg* as a revision list ordering heuristic because the same heuristics are also used to guide variable selection, so the domain sizes and constraint weights are already being calculated and recorded. After experimenting with structured and random problems, the authors find improvements on structured problems, but no significant advantage to using revision list ordering heuristics on random problems. The authors hypothesized that random problems may lack the hard local subproblems found in the structured instances. Thus, depending on which DWOs are encountered first, different areas of the search space will be explored, whereas the presence of hard local subproblems will direct the solver to focus on these parts first.

2.1.6 Variable Ordering Heuristics

The order in which variables are considered for assignment can drastically change the size of the search space. By considering the variable with the fewest remaining values we can quickly detect if any variable has no valid assignments remaining (Russell & Norvig, 2003). There are several search heuristics which may be applied to reduce the problem space. *Minimum remaining values* heuristic, also called *dom* for smallest active domain size, suggests trying to find an assignment for the variables with the fewest possible values first. By trying to satisfy this variable first we can reduce the branching factor of the search tree. Furthermore, if any variable has zero remaining values in its domain, the search will halt immediately as no satisfying assignment can be made.

A second useful heuristic is the *degree* heuristic, which orders variables in descending order based on number of constraints they are involved in. Thus, variables involved in many constraints will be assigned first. This is analogous to having the person with the busiest schedule suggest a meeting time and then checking if that time works for the other people.

Variable selection heuristics are usually intended to “fail fast”; that is, detect failures as early in the search tree as possible. If we can detect failures near the top of the search tree, then we can explore a much smaller portion of the tree compared to detecting failures near the bottom of the tree. In addition to failing fast, another common technique is to reduce the branching factor of the tree. By instantiating variables with fewer domain values first, we can create a tree with fewer branches and thus fewer states to explore during search. Haralick and Elliott (1980) are frequently cited for introducing the often successful *dom* variable ordering, which selects variables in increasing order of active domain size. In this section, we explore some modern research on variable selection heuristics.

A popular improvement has been to combine variable ordering heuristics, such as *dom/deg* (Bessière & Régin, 1996). Whereas the *dom* variable ordering suggests the next variable to be the one with the smallest remaining domain size, and *deg* suggests the next variable be the one with the largest number of constraints, *dom/deg* selects the variable that minimizes the ratio of the two orderings. Bessière and Régin claim that *dom* is most useful when there are many constraints, and *deg* is most useful when the constraint graph is sparse, so by combining the two they aim to find a better heuristic than either alone.

Alternatively, a single heuristic can be combined over a group of variables. (Bessière, Chmeiss, & Sais, 2001) propose the use of variable selection heuristics based on variables and their neighbours, rather than by considering each variable in isolation. The selection heuristic is defined recursively as:

$$H^{\circledcirc}_{(0,\alpha)}(x_i) = \alpha(x_i),$$

and,

$$H^{\circledcirc}_{(\kappa,\alpha)}(x_i) = \frac{\sum_{x_j \in \Gamma(x_i)} (\alpha(x_i) \circledcirc H^{\circledcirc}_{(\kappa-1,\alpha)}(x_j))}{|\Gamma(x_i)|^2},$$

where $\alpha(x_i)$ is some property of a variable such as *dom* or *dom/deg*, κ is the size neighbourhood to consider ($\kappa = 0$ corresponds to “standard” *dom* or *dom/deg*), and $\circledcirc$ is some method of combining values such as $+$ or $\times$. Thus, a property of a variable and its neighbours can be combined to create a value based on the local structure of the CSP.

Search experience can be used to inform variable selection, as is done with *wdeg* (Boussemart et al., 2004). Whereas *ddeg* selects the variable with the highest number of constraints involving unassigned variables, *wdeg* selects the variable with the highest sum of constraint weights involving unassigned variables. All weights are initially 1, and a constraint's weight is incremented whenever revising it causes a DWO. Thus, the hope is that constraints that frequently cause DWOs will be evaluated earlier in the search tree. *wdeg* can be combined with *dom* to create *dom/wdeg* in the same manner as combining *dom* and *ddeg* to produce *dom/ddeg*.

However, reweighting constraints based on DWOs is highly dependent on the initial search order and does not give any sort of global information (R. Wallace & Grimes, 2008). An alternative method is to preprocess the problem with random probing, using a random variable ordering for a fixed number of probes, with each probe consisting of a fixed number of assignments. Whenever a DWO is encountered in the random probing, the associated constraint is incremented. This allows for a more global representation of which constraints are frequently causing wipeouts and where search efforts should be concentrated. R. Wallace and Grimes (2008) show that using random probing to calculate weights improves upon both *dom/ddeg* and *ddeg*. Furthermore, they explore the effects of varying the number of probes and the number of states explored in each probe, finding that generally more and larger probes are better. From the results, adding constraint weightings can improve search cost by as much as 25%. By comparing the search cost using the weights obtained from probing ("frozen weights"), and updating the weights during search, the authors concluded that freezing weights is usually slightly better, and sometimes markedly better for random problems, but frozen weights are consistently much worse for colouring problems.

Beck, Prosser, and Wallace (2004) present a new measure of variable ordering heuristics that, in addition to the well known fail-first policy, helps to explain why some heuristics are better than others. The authors define *promise* as "the ability to make choices that lead to a solution when one exists," which implies that promise only matters for solvable problems. Furthermore, heuristics should show greater promise on problems with many solutions. Promise is measured by taking the probability of choosing a value for a variable randomly that leads to a solution the search tree and summing the path products (where the path product is the product all the probabilities assigned to the values in a path). Therefore, paths that do not lead to solutions are given value 0, and paths that do lead to solutions are given value equal to the probability of (randomly) selecting each of the values involved. The authors point out that when no consistency maintenance (e.g. FC, AC) is done, promise is equivalent to solution density; however, the introduction of consistency means that bad branches of the search tree will be pruned, and promise will be higher than solution density. The authors propose that different variable ordering heuristics demonstrate different levels of promise, and that promise is inversely correlated with search effort. As expected, the results show that as tightness increases, promise decreases, and the heuristics with the best search performance (fewest constraint checks) exhibited the highest promise scores. In a follow-up paper Beck, Prosser, and Wallace (2005) propose a new measure of failing-first: mistake tree size, or the number of search nodes explored after making a wrong assignment . They determine empirically that ranking the heuristics based on the size of the mistake trees they produce aligns with the actual search effort, rather than using the depth at which failures are detected. Thus, the fail-first policy is not truly a measure of failing early in the search tree, but making small digressions from the correct path. This work helps to provide an answer as to why a heuristic performs well, in terms of basic features of the search process, such as mistake tree size.

Refalo (2004) presents a new search heuristic based on the "impact" of a variable, where the impact corresponds to the number of domain reductions a variable or value induces during propagation. The author defines P as the product of the size of the domains of all the variables:

$$P = \prod_{d_i \in D} |d_i|.$$

The impact of an assignment is then defined as

$$I(x_i = a) = 1 - \frac{P_{after}}{P_{before}},$$

where P_{after} and P_{before} are P before and after making the assignment. Therefore, assignments that cause more domain reductions are higher impact. It should be noted that this is heavily tied to the choice of propagator, so an assessment of the influence of various levels of propagation on the impact heuristic would be interesting. The author notes that computing all the impacts which are needed to make a variable selection is extremely costly; however, they have observed that the impact of an assignment is nearly constant regardless of location in the search tree, so we can assume that the impact of making an assignment is the average of the impacts from previous times we have made this same assignment. Cambazard and Jussien (2006) examine the use of domain reduction explanations as the root for determining impacts, extending the similar work done by Refalo (2004). This work provides several new formulations of impact. First, the authors define an explanation e as a set of constraints and assignments that lead to a value being removed from a domain, E is the set of all explanations, and E_i^{val} is the set of explanations for the removal of val from the domain of x_i. The authors note that the "impact" of an assignment is not just the number of domain reductions it induces (as defined by Refalo), but that an assignment also has an effect on the future assignments as several assignments together may lead to a domain reduction. Therefore, they measure the impact of an assignment by

$$I_0(x_i = a, x_j, val) = \sum_{e \in E_j^{val}, x_i = a \in e} \frac{1}{|e|}.$$

Thus, each value has an impact on each other value, and by summing the reciprocal of explanation length, the more short explanations a value is involved in, the higher its impact.

A broad survey of many of the modern variable ordering heuristics was done by Balafoutis and Stergiou (2008a), using Maintaining Generalized Arc Consistency (MGAC) and lexicographic value ordering. The authors point out that recent research on variable ordering heuristics typically presents results from a narrow region of the problem space and compares a new heuristic to old heuristics. Thus, their aim is to compare several modern heuristics to each other over a broad range of problems. There is no clear winner across all problem instances examined, and most of the solvers examined are fastest for some problems; however, some solvers presented do not win on any of the provided instances. It is not clear whether some solvers are systematically inferior to other solvers, or if this is just a result of the problem sets examined.

2.1.7 Value Ordering Heuristics

Once we have used a heuristic to select a variable, we can use a second heuristic to guide the choice of which value to select for the variable. One such method is the *min-conflicts* heuristic which chooses the value that will cause the fewest domain reductions in the remaining variables if propagating using Forward Checking. Historically, there has been less research on value ordering than variable ordering, for two reasons: first, value orderings are typically more computationally expensive, especially dynamic ones (certainly this is true of min-conflicts); and second, for any unsatisfiable problem instance all values must be tried regardless of order.

Lecoutre, Sais, and Vion (2007) present two value ordering heuristics: *max-conflicts* and *min-inverse*, which are inspired by the Jeroslow-Wang heuristic from the Boolean Satisfiability literature (Jeroslow & Wang, 1990). Max-conflicts, the reverse of the well known min-conflicts, can be seen as a strict adherence to the fail-first policy, which is well known to be effective in CSP search. Lecoutre et al. point out that the 2-way branching scheme used in MAC actually causes value orderings to affect the size of the search tree. The authors note that recent work on value orderings has typically used d-way branching and static variable orderings, so they compare min-conflicts to max-conflicts using MAC on random binary problems, comparing the effects of *dom/ddeg* vs *dom/wdeg* and d-way vs 2-way branching. When using *dom/ddeg* and d-way branching max-conflicts is between 10% and 30% slower than min-conflicts. However, the introduction of *dom/wdeg* and 2-way branching significantly improves both heuristics and makes their mean times roughly equal. It should be noted that both min and max conflicts are precomputed, so that the value orderings are static rather than the typical dynamic min-conflicts implementation. This shows that the difference in variable ordering and branching method can make the two opposite value orderings have equivalent performance. The new heuristic min-inverse is presented, which is calculated before solving the problem and applied statically. The position of value $x_i = a$ in min-inverse ordering is calculated by summing the number of supports for $x_i = a$ in all variables x_j, where there is a constraint between x_i and x_j and the number of supports in x_i for the supporting values in x_j. Values are ranked in ascending order.

The process of choosing a variable and a value can be combined into a single selection (choosing from all values of all variables). Zanarini and Pesant (2009) present new combined variable-value selection heuristics based on choosing values that occur in the greatest number of solutions to a constraint. The three heuristics presented are MaxSD, MinSC;MaxSD, and MinDom;MaxSD. MaxSD chooses the variable-value pair that has the greatest ratio of constraint solutions including that value to constraint solutions. The other two algorithms pre-prune the possible choices by first selecting the constraint with the fewest solutions, or the variable with the smallest domain, respectively. These reductions result in only having to explore the variables participating in the tightest constraint, or the constraints involving the variables with the minimum domain size, respectively.

2.1.8 Problem Hardness

When generating parametric problems, there is typically some parameter for controlling the difficulty of a given problem. Intuitively, larger problems will tend to be harder to solve. However, within a given size of problem there is usually some method to control difficulty. Additionally, it has been noted that there can be significant variance in the hardness of problems generated with exactly the same parametrization, so methods to control the variance can make it easier to predict problem hardness from the parametrization (Sabin & Freuder, 1994; Cheeseman, Kanefsky, & Taylor, 1991). For a fixed problem size, adjusting the other parameters (such as constraint density and tightness for random problems) will vary the problem from trivially easy to solve, to difficult, to trivially easy to prove there is no solution. The transition from easy to difficult and back to easy is typically very sudden and occupies a very small region of the problem space, with the hard to solve problems occurring in the middle of the transition. These problems are hard because there are either very few solutions, one unique solution, or significant work is required to prove that there is no solution. In this section, we examine problem difficulty in random CSPs and quasigroups with holes.

Gent, MacIntyre, Prosser, and Walsh (1995) introduced a metric of hardness for random CSPs, τ[1]. τ values near 1 correspond to a peak in CSP problem hardness, with values greater than 1 indicating that the problem is over-constrained; that is, it is increasingly likely that there will be no solution. Conversely, values below 1 indicate that the problem is under-constrained and it is increasingly likely that there are many solutions to the problem. Under-constrained problems are easy to solve because many of the possible variable assignments will lead to solutions. Furthermore, over-constrained problems are easy to solve because propagation will quickly reveal that no partial solutions can be extended to complete solutions. However, problems with τ values near 1 will have few, 1, or zero solutions, but this will not be apparent from the top of the search tree. Typically, there will be many near-solutions, and the search will extend almost all the way down the search tree before any inconsistencies will be discovered. This results in minimal pruning of the search tree and thus large explored areas relative to the size of the problem (number of variables and domain values).

QWH problems are particularly interesting because they exhibit a easy-hard-easy phase transition, and all the problem instances are satisfiable. The phase transition over these problems ranges from problems that obviously have one single solution, to problems where there are many partial solutions that fail to lead to a solution (but at least one correct solution), to problems with many unique solutions. Typically, phase transitions have been described as a transition from under-constrained problems with many solutions, to critically constrained problems with one solution, to over constrained problems with no solutions. For QWH, as well as other problems based on Latin squares, the hardness of a problem instance is determined by the size of the *backbone*. The backbone of a problem, coming from the 3SAT literature, is defined as the percentage of variables that have the same value in all possible solutions (Zhang, 2001). Achlioptas et al. (2000) link the notion of the hard region to a sharp transition in the size of the backbone. There is a very narrow range in the number of holes in the problem where the size of the backbone transitions from near 1

[1]In later works such as (Gent, MacIntyre, Prosser, & Walsh, 1996), this metric was renamed κ, which has since become the accepted name. However, later in this thesis, a statistical measure of agreement called κ is introduced. So, we will refer to the CSP hardness metric by its original name, τ, and the statistical measure of agreement by κ.

(all variables take one single values in all solutions) to near 0 (all variables take on multiple values in the solution space).

2.2 Metareasoning

Metareasoning is the task of reasoning about reasoning. There are several reasons why we may want to do this, such as: reasoning to understand failures (such as reviewing a chess game to understand what sequence of moves lead to a loss), assessing problem solving performance to predict how much time will be needed to find a solution, or comparing solving performance to choose a solving method. Good's "type II" rationality, maximizing utility taking into account deliberation cost, is the goal of much of the metareasoning literature (1971). Not only is it important to have an algorithm that can produce good results, it is desirable to have an algorithm that can correctly balance the trade-offs between solution quality and speed. In this section, we will discuss the literature on general metareasoning architectures, metareasoning within a CSP, metareasoning about a group of CSPs, estimating runtime, metareasoning in the planning literature, and automatic configuration of parametric algorithms.

2.2.1 Reasoning About Reasoning

Rational Agent with Limited Performance Hardware (RALPH) is an agent with *limited rationality* proposed by Russell (1991). Limited rationality is defined as behaviour that maximizes utility in a changing environment with limited computational resources. Russell notes the importance of accounting for the costs of decision making as computational speed and memory are always limited. The author notes that an ideal agent would implement a function f_{opt}, which maps each situation to the best possible outcome. However, in practice, the number of steps required to compute the optimal action, or the space required to store all possible best actions, make the implementation of f_{opt} impossible. Thus, the author suggests pursuing "bounded optimality", or generating the best possible behaviour given the limitations of the hardware. Furthermore, the "best behaviour" is not limited to a single action, but rather refers to all actions taken in all states in which the agent is to operate. Finally, as the agent may operate outside of the designers expected domain, an optimal agent will learn most quickly the best actions to take. Russell notes that for the agent to make the best choices, it must perform valuable computations quickly, where valuable computations are ones that increase the utility of the actions to be performed, i.e. select better actions. As there are many types of information we may want to represent, a knowledge base of six types of information is proposed along with four different execution architectures (EA) to make use of the knowledge. The EAs overlap in the types of information they make use of, so they jointly share one knowledge base. At any given point in time, the EA that has the highest value computation available will execute until the time costs exceed the value of the computation.

A three tiered model for metareasoning agents has been proposed by Cox and Raja (2008). The authors extend the two-tiered agent action/perception cycle by adding a meta-level, responsible for monitoring and controlling the reasoning level. Four different types of metareasoning are discussed. First, *meta-level control* can be used to determine when to stop reasoning. For example, in planning there is a trade-off between refining the current plan and acting out the current plan, which can be decided at the meta-level. Second, *introspective monitoring* can be used to study and improve the reasoning at the object level. For instance, a classification system may receive feedback that a classification was incorrect. Then the systems classification model,and the problematic instance could be investigated at the meta-level to understand the failure and improve the classifier. Third, *Distributed Metareasoning* is the multi-agent task of coordinating metareasoning. For example, multiple agents each performing meta-level control, may need to jointly negotiate when to stop reasoning about the current task and what task to reason about next. Fourth, the authors describe *Models of Self* as a type of metareasoning in which an agent has an awareness of itself, learning from its actions or choosing actions based on its strengths and weaknesses.

However, some researchers, including Morbini and Schubert (2008) reject the highly cited "Ground level, Object level, Meta-Level" structure proposed by Cox and Raja for several reasons. First, the authors take issue with a three tiered approach, claiming there could be an arbitrary number of tiers providing control, or a single self-modifying planner. Additionally, the goals selected at the meta-level may be ground level goals (e.g. go to a location, perform an action) or they may be reasoning tasks (e.g. prove x is true, find a hypothesis accounting for a set of facts). Thus, the authors argue that the same controlling system, or metareasoner, may be controlling at multiple levels. Additionally, if there are known reasoning techniques used to solve a type of problem, then they become action-like and the distinction between reasoning and acting becomes blurred. Thus, the authors propose that the difference between metareasoning and reasoning is not an explicit structural one, but rather depends only on the syntax and semantics of the logic being used. However, it should be noted that neither of the models suggested by Cox and Raja (2008), and Morbini and Schubert (2008) precludes the other; metareasoning may be performed in different ways for different tasks.

Ulam, Jones, and Goel (2008) compare three different methods of training an AI player to effectively defend its cities from attackers in the game FreeCiv. The authors contrast four different methods: a static strategy that does not learn (the norm for most games), a model-based metareasoning agent, a reinforcement learning agent, and a hybrid metareasoning reinforcement learning agent. The model-based agent decomposes the city defence task into several parts and tries to assign blame to one of the subtasks after a failure. Once a subtask is determined to be the reason for failure, a library of possible modifications to the subtask is consulted. The obvious downside of this method is that a large amount of knowledge engineering is required to decompose the task into subtasks and to include adaption strategies for each task failure. By contrast, the reinforcement learning method does not require explicit instructions for how to fix a failure. The authors combine the two methods by decomposing the main task into several subtasks, as with the model based approach, and learn each subtask as an independent reinforcement learning task. The authors show the hybrid method has fewer failures and learns an appropriate strategy faster than either of the other learning methods.

In another work studying game playing, Jones and Goel (2009) use metareasoning to "repair" a classifier after misclassifications. They present a game playing AI which makes choices in FreeCiv, based on classifying possible moves. Later empirical verification of the possible moves enables the AI to determine if the previous classification was correct or not. The authors note that their self diagnosis and repair scheme enables the game playing AI to make 52% fewer classification errors after approximately 50 games have been played.

When discussing agents that can assess their thought processes, comparisons to humans are a natural extension. Gordon, Hobbs, and Cox (2008) discuss the application of anthropomorphism to metareasoning agents. By attributing human qualities to agents we can both create an internal model of an agents reasoning, which can be used to alter the agents reasoning via metareasoning, and it becomes easier for humans to understand, control, and relate to an agent that applies the same sort of metareasoning techniques found in humans. The authors present memory axioms for how information is retrieved, forgotten, remembered, repressed, and so on. By monitoring and controlling an agent's reasoning in such a humanistic manner, direct comparisons can be made to research in human common sense reasoning.

2.2.2 Reasoning Within a Single CSP Problem

One application of metareasoning to constraint satisfaction is to learn from the process of solving a problem in order to adjust how the remainder of the problem is solved. This could include previously discussed methods such as constraint learning or dynamic constraint weights.

El Sakkout, Wallace, and Richards (1996) demonstrate a method of choosing to apply AC or FC on a per arc basis, based on whether or not AC processing of the arc can cause more domain reductions than FC. There is some overhead in assessing all of the arcs and maintaining whether AC or FC should be applied to each, but the authors show that the reduction in wasted applications of AC outweighs the costs of maintaining the arc status. Additionally, adapting the propagation technique within a single CSP instance can be used to overcome a poor initial choice. The authors use AC-4 and point out that the speed advantage of FC comes primarily from recording fewer operations, as AC-4 maintains support counters for all domain values. As such, when backtracking occurs, all of the previously set support counters must be reset to the correct values after backtracking. Because FC maintains far fewer counters, the setting and resetting process is much faster. The authors define an *anti-functional arc* as one that has a support complement ≤ 1 for all values, where the support complement of a value is the difference between the domain size and the number of supporting values. The binary disequality constraint is an example of an anti-functional constraint. The authors then show that AC will not produce any additional pruning over FC on anti-functional arcs. Thus, arcs are processed with AC unless they are detected to be anti-functional, in which case they will be processed with FC. Detection of anti-functionality is quite efficient as the number of supports for each value are already stored by AC-4. The authors were able to show that runtimes were reduced by up to 60% over AC-4 when selectively applying FC to anti-functional arcs.

Another technique used to control the level of consistency applied is to monitor the number of DWOs (Stergiou, 2008). This work was motivated by the realization that when solving structured problems using MAC+*dom/wdeg*, constraints that cause domain wipe outs (DWOs) are often revised in clusters. That is, if a constraint caused a domain wipe out, then it is probable that it will cause a domain wipe out again soon. Thus, if DWO is found, then the author suggests applying higher levels of consistency to that constraint for the next several revisions, in the hopes of causing another DWO. Frequent discovery of DWOs will cause the constraint weights of *dom/wdeg* to be adjusted, thus causing the search to focus on the difficult parts of the problem first. The author notes that this technique is useful only on structured problems that have both hard and easy regions and points out that the methods are not useful for random or uniformly difficult problems. Therefore, it would be advantageous to be able to recognize problems for which this technique should or should not be applied.

2.2.3 Reasoning About a Group of CSPs

Several researchers have investigated methods for identifying a promising solving strategy for a group of like CSPs. One such method is to identify variables in a CSP based on the structure of their local neighbourhood in order to find similar variables in other CSPs (Day, 1992). Knowledge of variable similarity can then be exploited to inform value ordering in future CSPs. Thus, value orderings that perform poorly in one CSP will be avoided in similar future CSPs. But, by limiting the application to value ordering, rather than constraint learning, we can ensure that mistakes will not lead to missed solutions, but rather just slower solutions. Variables are identified in Day's classification by their domain range, currently assigned value, and the types of constraints in a limited local network. The author provides two example CSPs and demonstrates how the number of search node expansions are reduced in the second problem by following advice generated from solving the first problem.

MULTI-TAC, a Multi-Tactic Analytic Compiler for solving CSP problems is intended for situations where many problems of a similar class are to be solved (Minton, 1996). As such, it learns a solving method tuned to the specific problem distribution it is trained for and thus does not perform per-instance configuration. MULTI-TAC learns rules that match given heuristics, such as choosing the most constrained variable first by generating possible rules and checking if the rules match the heuristics. The benefit of such a technique is that there may be simple rules that agree with the heuristic with high probability and the heuristics may be expensive to compute directly, as is the case with the "most constrained variable" heuristic. The candidate rules are generated analytically through analysis of the constraints of the given problems as well as inductively by randomly combining features of the input problems. As one would assume, many of the generated rules are very poor guides for variable and value selection, so the author suggests discarding all rules that disagree with the heuristic more often than they agree. After a set of usually useful rules are gathered, an iterative beam search is done to choose the best CSP solver parametrization for the training set. Minton compared MULTI-TAC to algorithms hand-coded by humans on several problem distributions and found that MULTI-TAC was the best performing algorithm on one distribution and performed on par with the best human coded algorithm on the other distributions. Further, MULTI-TAC is compared to three non-tuned algorithms, GSAT, Forward Checking, and Tableau, and shown to be significantly faster than each.

In addition to devising an algorithm appropriate to the problem, M. Wallace and Schimpf (2002) investigate how problem formulation is as much a part of the problem solving process as algorithm configuration, highlighting the multitude of variable and constraint formulations that are possible. Furthermore, they note that problem formulations can be local; that is, one region of the problem may be described in terms of symbolic values with binary constraints and another region may have numeric values with integer constraints. This leads to an extremely large number of possible problem formulations, some more suited to solving with particular algorithms than others. With respect to CSP solver algorithm development, a common methodology is to view the solver as a series of operations (i.e. variable selection, value selection, propagation, backtracking), each of which is selected from a set of possible ways of performing the task (for example, variable selection can be done using any of the heuristics described in Section 2.1.6). Thus the task of configuring a solver algorithm is choosing appropriate assignments to each of the operations. The authors note that several recent hybrid algorithms could not have been developed in this way as they use new operations that are themselves hybrids of other operations. For example, Forward Checking and Arc Consistency are well known propagators, but recent work has developed methods that limit AC or extend FC, or apply AC and FC selectively to each constraint. These hybrid operators cannot be discovered by combining AC and FC; lower level building blocks are needed to describe them. Thus, the authors conclude that in addition to finding new ways to combine algorithms, we must also devise new ways of describing the parts of each algorithm such that they can be recombined in new ways.

Gebruers, Hnich, Bridge, and Freuder (2005) investigate the use of case based reasoning to predict fast performing CSP solver configurations for the social golfer problem. Like M. Wallace and Schimpf (2002), the authors consider not only all the parameters of the algorithm, but also the possible configurations of the problem. To clarify, most problems can be formulated in more than one way and the authors consider the set of possible formulations as another parameter to configure. The authors use a k-nearest neighbours approach to classify problem instances and contrast with the use of C4.5, the best average solver, an oracle solver, a randomly selected solver, and a weighted random selection (weighted based on frequency of being the correct choice). The authors found k-nearest neighbours to have slightly higher prediction accuracy than the other selection techniques and significantly lower total execution time for the chosen solvers. These results suggest that KNN is choosing correctly on more instances than the other classifiers, but more importantly, it is making more correct decisions on the most important instances.

Preprocessing can also be used to determine the propagation technique best suited to each individual constraint in a CSP, as was done by Stamatatos and Stergiou (2009). The reasoning for this is that some constraints will induce higher levels of pruning in a problem whereas other constraints offer little pruning, even if significant work is done enforcing consistency. The typical tradeoff in propagation techniques is between speed and power, and the aim of this work is to detect when more powerful techniques do or do not pay off. The authors preprocess by using random probing. Random probing is done by applying a search algorithm to a problem with a random variable ordering for a fixed number of steps. The authors employ a *staged propagator*, a sequence of progressively more powerful propagators, during the random probing and record which propagator was used, how many fruitful revisions it caused (a revision is fruitful if at least one value is removed), and the number of value deletions. By applying more powerful propagators in sequence, one can determine if a more powerful propagator is actually doing any more useful work beyond the earlier propagators. Because the staged propagator successively applies stronger propagators it is only used as a preprocessing step, as it incurs many redundant constraint checks that we would not want to perform during search. An obvious question is whether the propagation statistics gathered during random probing really match the propagation performance during a complete search. For each of the propagators used, the authors apply a clustering algorithm to group the performance on the constraints into three categories based on the number of fruitful revisions induced. They then show that for a selection of eight structured problems (two RLFAP, one driver (Long & Fox, 2003), one quasigroup completion problem, two quasigroups with holes, and two other problems), the data points from a full search were placed into the same clusters as with the random probing with between 58% and 95% accuracy. Furthermore, the accuracy of clustering constraints that are least likely to cause fruitful revisions was greater than or equal to 72%, indicating that detection of non-fruitful constraints is more accurate than more fruitful constraints.

The Learning Propagators through Probing (LPP) algorithm presented by Stamatatos and Stergiou (2009) consists of a decision tree based on information about which cluster a given constraint falls into and general performance information about the propagators gathered through the random probing. Unfortunately, there is no algorithm to construct the decision policy; it was hand tuned by the authors. The authors present a comparison of LPP with MAC and $H_{12}^{\vee}$ (another recent algorithm adapted from Stergiou (2008)) over problems from C. Lecoutre's web page[2] (RLFAP, graph colouring, haystacks, QCP, QWH, forced random problems). Lexicographic value ordering was used and the *dom/wdeg* variable ordering. Restarts were used with the first restart after 10 backtracks, the number of backtracks between restarts growing by 1.5X each time, and a timeout of 2 hours. The authors found that MAC outperforms LPP on the random problems, and they cite the lack of structure being the cause. On the structured problems, and particularly the larger ones, LPP is often faster than MAC by a wide margin, in one case completing a problem in 2 seconds where MAC timed out at 2 hours.

Hyper-heuristics, or heuristics to choose heuristics, are introduced by Bittle and Fox in order to make smarter decisions about what heuristic to use when solving a CSP (Bittle & Fox, 2009). The authors refer to the set of values needed by a heuristic to make a choice as *textures*. For example, the number of remaining values in each domain forms a texture used by *dom* to select the smallest domain size. The authors use an extended version of the SOAR cognitive architecture (Laird, Newell, & Rosenbloom, 1987) to build the hyper-heuristics. From the details provided, it seems that whenever SOAR does not have information about which heuristic to use (i.e, an unknown state), it does some amount of look-ahead to assess the value of all possible choices, preferring choices that result in fewer constraint checks. The authors compare hyper-heuristics to a benchmark solver using the *dom+deg* variable ordering and least constraining value (or min-conflicts) value ordering. As problem size increases, the hyper-heuristic solver is shown to improve relative to the benchmark solver.

Pulina and Tacchella (2009) present a method for learning to choose an appropriate Quantified Boolean Formula (QBF) solver for a given problem. As their work focuses on syntactic cues from QBF, the underlying problem domain is unimportant. This work chooses a solver based on problem attributes. If the solver fails to solve the problem in the specified time limit, an alternate solver is attempted. If the alternate solver succeeds, the instance is used to update the solvers' model. The authors make use of many problem features such as the number of variables, the number of clauses, the number of quantifier alternations, as well as the ratios and products of the basic features. Computing a total of 141 features takes on average 0.04s. The authors investigate the use of four different methods of training to distinguish problem instances: Decision Trees, Decision Rules, Logistic Regression, and Nearest Neighbour. Furthermore, the authors examine the choice

[2]`http://www.cril.univ-artois.fr/~lecoutre/benchmarks.html`

between two solvers, eight solvers, and 16 solvers. The authors note that as the number of possible solvers decreases, the odds of choosing the best available solver increase; however, as the number of possible solvers increases, the value of the best available solver is expected to increase. Their experimentation reveals that choosing among 8 solvers provided the better performance than 2 or 16.

Furthermore, Pulina and Tacchella (2009) note that, within the datasets they consider, a solving algorithm will tend to solve a problem within a few seconds or fail to solve the problem within several minutes, so switching solving methods after a few seconds elapse may be a promising strategy. If switching to a new technique solves the problem quickly, then the problem instance is used to update the algorithm selection model. The authors investigate multiple techniques for determining how much time to allow for each algorithm and in what order to apply the algorithms. The authors measure the performance in terms of the number of problems solved and total time taken to solve the problems, including retraining from instances where the initial assessment was incorrect.

The use of learning to guide the variable selection process while solving a CSP has been studied by Y. Xu, Stern, and Samulowitz (2009). In contrast to many other studies, that have examined the process of selecting a solving method for a given problem, the authors examined the task of learning a variable selection scheme, which is dependent on the current problem or subproblem. The authors used a reinforcement learning approach, where the reward of finding a solution is inversely proportional to the time needed to find the solution. The authors experimented with a single problem instance and found that the learning algorithm was able to learn to interleave two variable selection techniques, resulting in solving times significantly faster than either algorithm alone. Further work is needed to determine how well this method generalizes when the algorithm is trained and tested on disjoint sets. Furthermore, the proposed reinforcement learning method may be extremely slow when the search space is large.

Epstein et al. (2005) aim to automatically produce a propagation policy to suit a given class of problems. The authors explore a variety of propagation techniques including both Forward Checking and Arc Consistency as well as numerous methods intended to fill the space between the two, either by extending the “reach” of Forward Checking or limiting Arc Consistency. The propagation policy developed consists of: a decision as to which preprocessing methods to use, whether to use FC or AC or an intermediate method, which control parameters to use for the intermediate methods, and whether to switch propagation methods and at what depth in the search tree to switch. This work focuses on propagation methods for a specific class of problem. Thus, it is important that the training and testing sets be uniform. As all of the meta-level computation is done beforehand, the solver is not adapted to each individual problem; rather, the method best suited to the training set is chosen as the optimal method. In cases where many similar problems must be solved, this would be a good approach; however, the assumption that new problems are similar to the training problems limits the generality of the method.

In a later work, Epstein (2009) describes the problem of choosing a problem solving method on a per-problem basis in terms of *descriptives* and *advisors*. Descriptives provide some sort of information about the problem, and advisors make use of this information to suggest a next action. Descriptives may be general properties of the problem, such as the number of variables or the average domain size. Alternatively, descriptives may provide some sort of local information, such as clique detection, or may be related to the current state of the CSP solver, such as detecting when backtracking occurs. Advisors make use of one or more descriptives to provide advice about what action should be taken. Advisors could be used for propagation (i.e. in what order should constraints be propagated? what type of propagation should be enforced?), or could similarly be used for variable or value selection.

Epstein's (1992) method is implemented using "For the Right Reasons" (FORR) , and three tiers of advisors: tier 1 advisors always provide quick and correct advice, although they do not all provide advice in all situations; tier 2 advisors are triggered by some event and are given are given a time limit to determine an action; tier 3 advisors are used in a weighted voting system to determine which action to take. The FORR based implementation for solving CSPs is called the Adaptive Constraint Engine (ACE). FORR/ACE learns from completed CSP traces, treating decisions that lead to the solution as positive training instances and decisions leading to digressions (failed search branches) as negative training instances. Training instances are considered sequentially and advisor weights are adjusted after each problem in the training set. A *benchmark advisor* gives random advice and any advisors shown to give worse advice than the benchmark are removed. This process not only increases the accuracy of the remaining advisors, but reduces computation time as any descriptors used by the removed advisors can also be removed. Epstein presents experiments training on 30 problems from the class $\langle 50, 10, 0.38, 0.2\rangle$ and testing on 50 problems from the same class. The best single heuristic considered solves these problems with an average search tree size of 30,024.66, whereas ACE produces an average search tree size of 8,559.66. Although this is a large reduction in search tree size, the process of consulting many advisors is time consuming and may not result in a faster time to solve the problem.

Several difficulties with learning to solve CSPs are highlighted by Epstein and Petrovic (2007). First, ACE uses reinforcement learning with ⟨search state, decision⟩ pairs; all variable orderings will lead to a solution, if one exists, and all variable orderings can generate error free searches, if the right values are chosen. To address this, in this work, the authors consider a variable selection incorrect if the subsequent value selection does not lead to a solution. A second difficulty is that it can be hard to train and test on similar instances; even problems generated with the same $\langle n, m, p_1, p_2\rangle$ exhibit large variation. It is hard to judge the difficulty of a class of problems from some instances, for exactly the opposite reason: problems drawn from the hard region may be easy, moderately hard, or very hard. A third difficulty is that errors occur in search whenever a wrong value assignment is made. "Good" solvers will detect an incorrect assignment quickly and backtrack. However, assessing how quickly a solver detects an error and backtracks requires a comparison to the other possible solvers or variable/value orderings, making this a difficult task. Errors may not be detected quickly in that it may only be clear that a mistake was made after several more assignments, at which point it becomes unclear what the true error was, as several variables now jointly cause conflict, and it may be the case that a different variable ordering would have caused less search.

Petrovic and Epstein (2008) present a method of learning a weighting among heuristics to solve CSPs with fewer variable and value selections. The authors used ACE in their work and used MAC-3 as their search and inference method. The authors show that both well known heuristics and their opposites (such as min domain size, and the inverse max domain size) can be highly effective, depending on the problem class. ACE uses a weighted combination of heuristics to choose variables and values, which results in smaller search trees. The authors explore the use of random subsets of heuristics to speed learning. As noted, positive training instances are derived from successful searches so when a set of advisors is dominated by poor advisors no positive training instances will be found. The goal is to use subsets of advisors to increase the odds of a subset having mostly good advisors. If a subset is found to contain mostly good advisors, then a solution can be found and the weights of the advisors can be updated. The authors show that the learning phase of repeatedly randomly selecting advisor subsets for learning advisor weights is significantly faster than using the entire set of advisors and the resulting weighted heuristics are significantly more effective in the testing phase.

2.2.4 Runtime Estimation

In addition to predicting an appropriate solving method, several works have focused on estimating the amount of time required to solve a problem. One such work by Horvitz et al. (2001) aimed to predict the runtime of satisfiability problems encoded as both CSP and SAT. The authors used a Bayesian model to identify the most important problem attributes, from which they construct a decision tree to predict whether an instance will have runtime above or below the median of the dataset. One application the authors suggest is the control of dynamic restart policies in solving algorithms. Over four QWH datasets classification accuracy ranged from 60% to 96%.

Another work in the SAT domain by Haim and Walsh (2008) presented a method of predicting runtime based on early performance on a given problem. Two of the issues cited are that clause learning is continually changing the problem that predictions are being made about and that restarts lead to extremely different areas of the search space, so information gathered about a problem may not hold true after a restart. As predictions are being made at runtime, only quickly collected attributes are used including features of the problem and the search behaviour. The authors find that, for search without restarts, being further in the search tree (measured by the number of backtracks) results in an increase of approximately 5% to 15% in prediction accuracy. Furthermore, when using restarts, including attributes about previous restarts results in an approximately 5% increase in prediction accuracy, indicating that some attributes from previous restarts may still be informative. However, it should be qualified that these percentages were inferred from the authors' figures as specific numeric data was not presented.

SATzilla, the competition-winning SAT solver, makes use of empirical hardness models to determine which algorithms to apply to each problem (L. Xu et al., 2008). The SATzilla method is as follows: first, two presolvers are applied sequentially in order to solve exceptionally easy problems; second, features of the problem are computed; third, using empirical hardness models, the solver expected to be fastest is applied. If the process of computing the problem features reached a predefined timeout, the solver previously determined to have best average performance is applied. If the predicted algorithm fails or reaches a timeout, then the second best solver is applied and so on until time runs out. Later work by L. Xu, Hutter, Hoos, and Leyton-Brown (2009) added a predictive model to determine if the feature computation will time out and, if so, applies the backup solver without attempting to compute the problem features. The authors use both a set of basic problem features and pairwise products of features to build a regression model. Furthermore, the authors use a hierarchical hardness model which first determines the probability of a problem being unsatisfiable or satisfiable and then, using two empirical hardness models (one for satisfiable problems, and another for unsatisfiable problems), combines the predictions of both models based on the likelihood of the problem being satisfiable or not.

Zheng and Horsch (2005) present a metareasoning constraint optimization solver. Constraint optimization is an iterative process of finding successively better solutions to a problem. Each solution has a potentially unique value, as compared to CSP problems wherein all solutions are equally valuable. In this work, the authors weigh the relative values of solution quality and the time taken to solve the COP. Whenever a new best solution is found, the metareasoner predicts the likelihood of finding the next solution quickly. If it is expected that a better solution will not be found quickly then the search process is halted. Alternatively, if a better solution is expected to be found quickly, the search process continues. The authors show that when there is a cost associated with the time taken to solve the CSP, metareasoning can be used to improve the total value encompassing solution quality and time cost.

2.2.5 Learning in the Planning Domain

As with constraint satisfaction, several researchers have investigated the use of learning to improve plan making. Estlin and Mooney (1997) discuss the SCOPE learning system for improving planning.

The authors note that SCOPE (Search Control Optimization of Planning through Experience) is able to increase both the efficiency of generating plans as well as the quality of the resulting plans. SCOPE learns from selection-decision examples, which are planning subgoals and the corresponding action chosen to solve the subgoal. In addition to the selection-decision examples, a proof tree is generated to determine which selection-decision examples are positive or negative examples. SCOPE learns control rules inductively using FOIL (Quinlan, 1990). Interestingly, the authors show that SCOPE can be trained to maximize various plan qualities based on the training input given, rather than some other control parameter. For example, to train SCOPE to produce optimal plans, the authors use a depth-first iterative deepening to generate plans of optimal length and then use these examples for training.

Huang, Selman, and Kautz (2000) present a strategy for encoding domain specific control instructions as additional constraints in planning problems. One of the clear advantages of this strategy is that the control information is not specific to the solver used. Thus, learned information can be transferred from one solver to another. The problem domain presented is the logistic domain of routing packages through an air mail and delivery truck system in order to minimize the time to deliver all packages. There are several actions including loading, flying, unloading, and so on, and an optimal sequence must be selected. The learning system will learn new rules such as “a plane cannot fly to a new city if it is loaded with a package to be delivered in the current city”. This rule is not explicitly encoded in the base rules for the problem, but this new rule holds true for all optimal solutions in the training set. The authors warn that using small training sets could result in learning overly specific rules that will not generalize to the testing set.

The process of learning to generate STRIPS/ADL plans from random training instances is presented by Fern, Yoon, and Givan (2004). By training on randomly generated problems from a particular domain, the authors aim to improve performance on “real” problems drawn from the same domain. Problem instances are generated for a given planning domain by creating a random initial state and performing a random walk of length n in the problem space. The “goal” state is then set to be some subset of the features of the state at the end of the random walk (i.e., not all features are to be considered important). The planner is trained on instances of size n until some predetermined level of success is achieved, at which point n is increased and harder problems are used for training. The authors note that randomized instances are not necessarily like traditional instances and, as such, orthogonal techniques should be used to augment the learned knowledge. The benefit of this technique is that the planner can learn from randomly generated instances and small randomly generated instances can be used to bootstrap the training on larger instances.

Tsoumakas, Vrakas, Bassiliades, and Vlahavas (2004) investigate the use of KNN for predicting which planning algorithm configuration will be most effective. The planning algorithm used, HAP, has 7 parameters each with several values, resulting in a total of 864 configurations. The planning problems investigated consist of 450 problem instances from 15 problem domains (30 problems from each domain). Finally, the authors aim to predict a parametrization that maximizes a user preference based on time to generate a plan and the length of the resulting plan. Thus, the user is able to specify their preference between short planning phases and short plans. One possible problem with using KNN, and a Euclidean distance metric, is that the relative importance of attributes is lost. Furthermore, redundant attributes and irrelevant attributes are not ignored, as they are in some other learning methods (such as decision trees). Training data are collected by running all parametrizations of the solver on all problems. However, this method may be impractical as the number of problems and solver configurations are increased. To evaluate the predictive solver, the authors compare to the configuration with the best average performance as well as to an oracle that chooses the best configuration for each problem. The authors found the performance of the predictive solver to be approximately half way between their two benchmarks for three different user preference schemes, indicating both that a significant improvement can be make over the best average solver and that there is room to improve towards the oracle solver.

2.2.6 Automated Tuning of Parametric Algorithms

Hutter, Hamadi, Hoos, and Leyton-Brown (2006) provide a method of predicting run times of randomized and parametric SAT algorithms. Their work is novel in that previous research has involved generating empirical hardness models for complete deterministic search, but not incomplete and random search. Based on their method of predicting runtimes of parametric algorithms, they also present a method of per-instance automatic algorithm configuration which is shown to outperform the best single algorithm over a problem distribution by an order of magnitude. The authors begin with a set of 43 problem features and consider basis functions consisting of the raw features as well as complex functions over the features using forward selection to choose a feature set. Ridge regression is used to determine a linear function, which is used for assessing new problem instances.

In a follow up work, Hutter, Hoos, and Stützle (2007) introduce a method of configuring algorithms by using local search to determine optimal parameter values. The authors note that parametric algorithms are often hand tuned, resulting in stopping at local maxima and missed opportunities for improved algorithms. Thus, they apply iterated local search (ILS) to tune algorithms. ILS is an iterative method that randomly perturbs the current solution, performs a local search, and has a method for determining if the new local maxima is better than the last. The goal of their work is to examine a small set of problem instances in order to determine the parametrization expected to have the best performance on a larger set of problems. In later work, Hutter, Hoos, Leyton-Brown, and Stützle (2009) expand their local search for algorithm configuration, called paramILS. This work introduces a new feature to the algorithm configuration search, namely *adaptive capping*, whereby the assessment of a candidate configuration is terminated if it cannot be better than the best configuration so far. For example, if the average time to solve a problem is the metric being minimized and solving all remaining problems in the test set in zero time would result in an average time greater than the best solver so far, then it is not possible for the current solver to outperform the best so far and thus the current configuration should be terminated. Two variants of adaptive capping are considered. First, trajectory-preserving capping resets the "best configuration so far" after each random perturbation in the ILS. This allows for examination of configurations perturbed into initially poor states. A second variant, called aggressive capping, bounds the performance of all configurations to be within a constant factor of the globally best variant so far. Finally, the authors note that paramILS is itself a parametrized search algorithm and present results of using paramILS to configure itself. However, the self-configured variant is shown to be only marginally better than the standard version, which the authors attribute to having initially guessed a good parametrization for the standard configuration.

Chapter 3

Methodology

In this chapter, we provide an overview of our metareasoning CSP solver, as well as more detailed descriptions of the CSP solving algorithms, machine learning algorithms, and experimental data sets. As we have seen in previous literature, the choice of optimal CSP problem solving method is highly dependent on the type of problem or, within a family of problems, the specific problem configuration. In this work we aim to predict which CSP instances should be solved using which propagation method. We focused on deciding between solving a given problem using Arc Consistency, or Forward Checking, with all other solver parameters fixed (e.g. variable ordering, value ordering, backtracking technique). We used a supervised machine learning approach, generating our training data by solving a set of problems using both AC and FC. Experiments were performed with randomly generated CSPs (with no predetermined structure), small world graphs with random constraints on each edge, and quasigroups with holes. The CSP solver is coded in Java and was run on a computer using Ubuntu 10.04 with a 2.33 GHz Intel Core 2 Quad processor and 4 GB RAM.

3.1 Metareasoning Solver Overview

The metareasoning CSP solver consists of a CSP solver, a data set of solved problems to be used for training, a machine learning element to construct a learned model and the learned model or performance element to predict which solver to apply to which problem. Our metareasoning process builds a decision tree using the database of solved problems and, based on the attributes of the problem currently being solved, uses the learned model to classify the current problem.

The same methods could also be used to choose variable or value selection heuristics, or other solver parameters. A completely configurable solver is possible too, with selection of heuristics, propagation technique, backtracking method, and constraint learning technique all chosen by a metareasoner. It is unknown whether all combinations of solving methods would be useful in some cases or if there are CSP solvers that would be clobbered by others.

The metareasoning process can be done online or offline and can be used to tailor the solving method to a specific problem or to choose a solving method for a class of problems. Some related works use a metareasoning technique to choose a solver appropriate to a given class of problems e.g., (Epstein et al., 2005), others configure the solver to each problem encountered e.g., (L. Xu et al., 2008). In the present work, we train the metareasoner to perform on a broad class of problems and calculate the cost of metareasoning about each problem.

3.2 CSP Solver

As we have seen in Chapter 2, the range of CSP solver configurations is quite large. It would be infeasible to experiment using all the solvers so we have chosen to use AC-3 and FC (refer to Section 2.1.5) for our experimentation because they are two of the most commonly sited propagation techniques. Previous works have debated the usefulness of AC and FC, citing areas of the problem space where one method clearly outperforms the other (Bessière & Régin, 1996; Nadel, 1990; Sabin & Freuder, 1994). Many of the newer propagation techniques are refinements of either AC or FC which reduce the total number for constraints examined, often at the expense of increased storage size. However, we are more interested in whether problem-specific algorithm selection can be done, and which problem attributes are indicative, than in designing a state of the art CSP solver. Future work should explore the use of additional solvers as well as more modern algorithms. Our general search algorithm is presented in Algorithm 1, Arc Consistency-3 propagation is presented in Algorithm 2, the Forward Checking propagation algorithm is presented in Algorithm 3, and the revise procedure used by both AC and FC is presented in Algorithm 4.

The task of propagation is to reduce the possible search space by pruning domain values that cannot lead to solutions. However, we must perform many constraint checks, not all of which will lead to domain reductions. If there will be many ineffective constraint checks, then time might be better spent exploring the search tree rather than pruning. Arc Consistency performs more constraint checks than Forward Checking each time it is called, whereas Forward Checking will explore more nodes in the search tree in a fixed amount of time.

The total cost of applying a solving method is best described by the time required to solve a problem using the given method. Although time is dependent on both implementation and hardware, it provides a complete measure of the cost of solving. By contrast, a solver with few constraint checks may take more time to solve than a solver with many constraint checks, once we consider the other costs of solving, such as node expansions and backtracks. The cost of solving could be modelled by counting the number of times each action is performed, such as a constraint check or a node expansion. However, modelling the total cost is then dependent on the relative speed of each action, and various solver implementations may have implemented each action differently. For example, in our implementations, the data structures used to store the queue of constraints are different between AC and FC. As such, there is a different overhead associated with a constraint check when using AC or FC. The overheads come from the costs of inserting and removing values from the data structures, as well as maintaining and checking additional data structures, such as finding the neighbouring variables to add new constraints to the processing queue after a domain reduction using AC. Finally, when comparing two algorithms, it is common to examine their average or worst case complexity, but in this work we aim to predict performance on specific problems, which is tied not only to the algorithm but also the implementation. Thus, we use time as our measure of performance.

3.3 Machine Learning

In this work we used a set of CSPs solved with both AC and FC as training data to learn to predict which solver is faster for a given problem. We used 10 fold cross validation in order to train and test on every instance in our dataset without training and testing on the same instances at the same time. It is important to not use the same instances in both the training and test sets, as this would give unusually high performance measures. Within a single fold, 10% of the data is used for testing and the remaining 90% is used for training. This process is repeated 10 times, selecting a different 10% for testing each time. After 10 iterations, or folds, we have tested on every data point, but at no time was the same data point used in both the training and testing set. By keeping the training and testing sets separate we are able to more accurately assess the performance of the algorithm on future problems.

There are machine learning algorithms for both classification and regression. Classification algorithms aim to produce a label for each instance, indicating the algorithm predicts the instance belongs to the given label. Regression algorithms are used to predict a numeric value, given a set of inputs. Our task of selecting an algorithm believed to be faster can be formulated as either regression or classification. We chose to consider a classification task rather than a regression task as we are only choosing between two methods. If a choice between many methods is to be made, then regression may be preferred.

3.3.1 Decision Trees

Decision trees (Quinlan, 1986) consist of a series of conditional tests to perform, beginning at the root of the tree, which lead to classifications at the leaves of the tree. The tests of a decision tree consist of checking an attribute and, in the case of numeric attributes, taking one action if the attribute is above a given value or another action of the attribute is below the given value. The "actions" are to either return a classification or perform another test. The depth, or number of attribute checks, in the decision tree depends on the "purity" of the data or how well splitting on an attribute splits the classification. Attributes are tested in order of decreasing information content.

If after testing all attributes the data is still not pure, as is the case with noisy data, the majority class is returned.

Two degenerate cases of decision trees have been shown to be useful. First, the ZeroR algorithm finds the class in the training set with the greatest number of occurrences and predicts each element in the test set to be of this class. This technique is analogous to guessing the class that has been seen most frequently. This can be effective for problems where the distribution among classes is very one sided. However, if the distribution is even, then ZeroR is equivalent to random guessing. ZeroR is equivalent to making a single leaf in a decision tree. OneR (Holte, 1993), the second degenerate case, is a single level decision tree splitting on one attribute. For each attribute in the data, a decision tree is made and compared to the decision trees from the other attributes. The decision tree with the lowest error rate is the one used. The OneR learning rule has been shown to be very effective and often competitive with much more complicated learning algorithms.

3.3.2 Other Learning Algorithms

We also considered using several other machine learning algorithms, including Naïve Bayes, neural networks, and k-nearest neighbours.

Naïve Bayes models assume all the attributes are conditionally independent, given the classification. The prediction model is as follows: given the attributes $x_1 \ldots x_n$ and the classification variable C, the probability distribution for C is

$$P(C|x_1 \ldots x_n) = \alpha P(C) \prod_i P(x_i|C)$$

(John & Langley, 1995). The probabilities for each attribute are observed from the training data set. The predicted classification is the class with the highest probability.

Neural networks are directed networks of nodes, which are called neurons. Neural networks aim to function like the networks of neurons in the human brain. Each neuron functions by taking a weighted average of its inputs (which are either active or inactive), applying a nonlinear function, and producing an output (again, either active or inactive). The output from one neuron is then used as the input to one or more other neurons (Russell & Norvig, 2003).

The k-nearest neighbours model claims that the classification for a given instance should be similar to the classification of similar instances. In order to normalize attributes of varying scales, we can replace attributes with the corresponding multiple of standard deviations away from the mean value. With all the attributes described in terms of standard deviations, we can simply use the Euclidean distance to determine the nearest neighbours. Once the nearest k neighbours to a query instance have been found, we can take a majority vote for the classification. Further discussion of the nearest neighbour learning algorithm can be found in Aha, Kibler, and Albert (1991).

3.3.3 Combining Learning Algorithms

In addition to the learning algorithms described above, there are several ways to combine multiple learning algorithms, such as Ensemble Learning and Boosting.

Ensemble learning involves using multiple classifiers and combining their results. For example, 10 different classifiers could be used on a given instance and the majority vote of the classifiers is used. Thus, k-nearest neighbours is an ensemble variant of the nearest neighbour algorithm, combining the first k nearest neighbours.

Boosting is a method of attempting to improve the correct classification rate by weighting the importance of each training instance. Initially, all training instances are weighted evenly and the chosen learning algorithm is used on the training data which produces one model. We then re-weight the training set by decreasing the weight of the correctly classified instances and increasing the weight of the incorrectly classified instances and train a second model. By increasing the weight of the instances that were initially misclassified, the second model will put a greater value on these instances. This process can be repeated an arbitrary number of times to produce many classifiers, each focused on areas where previous classifiers were lacking. Finally, after all the classifiers have been trained we collect a majority vote for classifying new instances.

3.3.4 Comparing Decision Trees With Other Learning Algorithms

We have chosen to use the j48 decision tree algorithm from the WEKA[1] toolkit in our work, based on C4.5 (Quinlan, 1993). We compared decision trees with the other methods described and found some advantages. First, all methods performed fairly well in terms of their prediction accuracy. However, the nearest neighbour and neural networks were significantly slower than Naïve Bayes and Decision trees. Decision trees also had the benefit of producing a visually intuitive tree for understanding the classification process. There is no particular reason why any other learning algorithm could not have been used in place of j48, but for the sake of tractability we chose not to further consider alternative learning algorithms.

3.3.5 Data Cleansing

In our experiments, we compare training on all the meaningful data we have collected versus training on a subset of the data. In particular, we are interested in training on the most important instances. We expect that there are some regions of the problem space that are clearly better solved with one method or the other, but that there are also boundary regions where there is little difference between the two methods. We expect that these regions are essentially noise because they will indicate that different solvers should be used for neighbouring problems, with extremely

[1]Available from `http://www.cs.waikato.ac.nz/ml/weka/`

similar properties, although the preferences will be very small. By removing all the data points where there is only a small difference in solving time, we are removing these noisy data points, and leaving behind the ones that more substantially favour one solver or the other.

A second technique we use to overcome the problem of noisy data is to weight the training and testing instances based on their importance. We assign the importance of an instance to be the difference in solving time between the two methods. Thus, instances that are solved much faster with one method than the other will have high importance and instances with little or no difference in solving time between methods will have low importance. After splitting the data into training and testing sets, each instance is multiplied by its importance, thereby creating clones in the dataset. The more times an instance is cloned in the training data, the more the decision tree will favour similar instances. Also, the more times an instance is cloned in the testing set, the larger the reward (penalty) for correctly (incorrectly) classifying it. Thus, the difference in runtime between solving algorithms is used as the misclassification cost when evaluating the learned model.

3.4 Metareasoning

In our application, the goal of metareasoning is to make informed decisions about what type of propagation to apply. Propagation, in turn, is reasoning about how to solve the combinatorial problem contained in the CSP. We apply our metareasoning as a post hoc analysis, in that we use a collection of solved problems as our testing set and determine which algorithm the metareasoner would choose to apply to each problem. This sort of analysis is sufficient because once a propagation method is chosen, being either AC or FC, the remaining time to solve the problem using either propagator will be exactly the same as the time to solve the problem with the chosen method without metareasoning. Therefore, the time to solve a given problem via metareasoning will be the sum of the time taken to metareason and the time to solve using either AC or FC.

3.5 Problem Space

For our experimentation we have chosen to consider three types of CSP. First, we consider random CSPs, generated using model B (refer to Section 2.1.2 and Table 2.1). Second, we used small world graphs with random constraints applied to each edge of the graph. This problem class is similar to the technique used by Gent et al. (2001) to add random constraints to structured quasigroup problems. Finally, we also used quasigroups with holes, which are a variant of the quasigroup completion problem in which all problems are guaranteed to have at least one solution.

3.5.1 Data Collection

Various methods of data sampling have been used in the CSP domain, such as uniform sampling, sampling problems from the hard region, or sampling particularly sparse or dense problems. We chose to use uniform sampling over a large problem space to avoid sampling biases, as we know various solving techniques perform better or worse in different regions of the problem space. We attempted to choose a broad enough sample of problems in order to show both AC and FC are valuable on some problems, as it is not possible to learn when to apply FC if all the training examples show AC to be superior. Because of our broad sampling, there is large variation in the time taken to solve different problems. As such, we have set a timeout of 30 minutes to limit the amount of effort applied to a single problem. This timeout ensures that we are able to collect a large number of data points in a reasonable amount of time.

We collected attributes of each problem as well as the time to solve the problem with each solver. Both solving techniques were applied to all problems in order to directly compare solving techniques on particular problems. The problem attributes we collected about each problem are as follows:

- **n** Number of variables.

- $\mathbf{p}_1$ Density of constraints. Defined as the number of edges in the constraint graph, e, divided by the maximum possible number of edges:
$$\frac{2 * e}{n(n-1)}. \tag{3.1}$$

- $\mathbf{p}_2$ Constraint tightness. Defined as the average fraction of disallowed domain pairs summed over all constraints. Defined as
$$\frac{disallowed}{c * m^2}, \tag{3.2}$$
where *disallowed* is the number of disallowed domain pairs over all constraints, c is the number of constraints, and m is the domain size.

- $\boldsymbol{\tau}$ A measure of problem hardness for random CSPs, defined as
$$\frac{n-1}{2} p_1 log_m(\frac{1}{1-p_2}). \tag{3.3}$$
Values near 1 correspond to the region of the problem space containing the hardest problems. Values less than 1 are associated with problems with multiple solutions. Values greater than 1 indicate that a problem will likely have zero solutions.

- **tightest constraint** The tightness of the tightest constraint. The maximum
$$\frac{disallowed}{m^2} \tag{3.4}$$

over all constraints, where *disallowed* is the number of disallowed domain value pairs, and m is the domain size, over all constraints.

- **loosest constraint** The tightness of the least tight constraint. The minimum
$$\frac{disallowed}{m^2} \tag{3.5}$$
over all constraints, where *disallowed* is the number of disallowed domain value pairs, and m is the domain size, over all constraints.
- **tightness variance** The variance over the range of tightness values for all constraints in a CSP.
- **average domain size** The average domain size over all variables in a CSP.
- **smallest domain size** The size of the smallest domain over all variables in a CSP.
- **largest domain size** The size of the largest domain over all variables in a CSP.
- **domain size variance** The variance over the range of domain size values for all variables in a CSP.
- **max degree** The number of edges connected to the variable with the most edges.
- **min degree** The number of edges connected to the variable with the least edges.
- **average degree** The average number of edges connected to a variable.
- **degree variance** The variance of the number of edges connected to a variable.
- **total number of edges** The total number of edges in the CSP.
- **acyclic** A Boolean value indicating whether the CSP constraint graph has cycles or not.
- **average path length** The average number of edges on the shortest path between all pairs of variables.
- **longest path** The longest path out of the shortest paths between all pairs of variables.
- **path length variance** the variance in length of the shortest paths between all pairs of variables.
- **graph width** An ordered graph consists of a graph G and an ordering d. G consists of a set of variables V and a set of edges E. The nodes connected to v in the constraint graph and preceding v in the ordering d are called the parents of v. The width of a node in an ordered graph is its number of parents. The width of an ordering of a graph is the greatest width of all the nodes in the ordering. The width of a graph is the minimum width over all possible orderings of the graph.

- **induced graph width** An induced graph is created by processing the an ordered graph, beginning with the last variable in the ordering and proceeding to the first. As each variable is processed all of its parents are connected. Induced graph width is then then graph width of the induced graph.

- **clustering coefficient 1** The ratio of triangles to connected triples in the graph. A group of three nodes in a graph form a triangle if they are fully connected. Three nodes form a connected triple if they are either fully connected, or there are two edges between the three of them.

- **clustering coefficient 2** For a given graph node x_i, the clustering coefficient C_{x_i} is defined as the p_1 over the subgraph consisting of the neighbourhood of x_i, $\Gamma(x_i)$. Clustering coefficient 2 is then the average C_{x_i} over all nodes in the graph. This metric was proposed by Watts and Strogatz (Watts & Strogatz, 1998).

3.5.2 Random Problems

The first class of problems we investigated are randomly generated problems. We describe random problems using four control parameters, n, m, p_1, p_2. n and m are the number of variables and the domain size of each variable. Random problems have no defined structure, but a control parameter, p_1, determines what fraction of the possible edges in the constraint graph are included. Values of p_1 range from 0 to 1, where $p_1 = 1$ corresponds to a complete graph and $p_1 = 0$ corresponds to a graph with no edges. A second control parameter, p_2, refers the fraction of value pairs in each constraint that are disallowed. Values of p_2 range from 0 to 1 where $p_2 = 0$ indicates that all pairs of tuples are allowed by each constraint and $p_2 = 1$ indicates that for all constraints, all pairs of tuples are disallowed.

We used random problems with the following configurations: n=[10,30,50], m=[10,30,50], $0 \leq p_1 \leq 1$, and $0 \leq p_2 \leq 1$, increasing p_1 in increments of 0.1 and p_2 in increments of 0.01. For tractability reasons we limited the number of configurations of n and m. There were a total of 9,000 problems.

Obviously, as p_2 increases, the probability of a problem having a solution decreases. There is a small chance when generating random problems that the problem will be insoluble, even if p_2 is not very large. Gent et al. (2001) introduce the concept of a flawed variable, namely one that does not have an allowable assignment given its neighbours. Problems that contain flawed variables are generally trivially insoluble; making the problem arc consistent will detect any flawed variables. Detection of a flawed variable indicates that the entire problem will have no solution. Gent et al. (2001) give the following probability that a problem generated using model B has a flawed variable:

$$1 - (1 - (1 - (1 - \binom{m^2 - m}{p_2 m^2 - m} / \binom{m^2}{p_2 m^2})^{p_1(n-1)})^m)^n.$$

This probability increases with n, p_1, and p_2, and inversely to m. Furthermore, Gent et al. (2001) introduce a method of generating flawless constraints to reduce the probability of large problems being trivially insoluble. The benefit of using flawless constraints is that CSPs using flawless constraints are not trivially insoluble for $p_2 < 0.5$, regardless of the other three parameters. However, for $p_2 \geq 0.5$, CSPs using flawless constraints tend to be trivially insoluble as $n \to \infty$. By contrast, the authors show that "flawed" (not flawless) problems tend to be insoluble for $p_2 \geq 1/m$ as $n \to \infty$. However, by design flawless constraints are initially arc consistent, which is a property that may or may not exist in "real" problems. As such, we have chosen to use "flawed" constraints in our experiments. Furthermore, we have investigated the impact of flawed constraints on our experimentation in table 3.1.

Table 3.1: List of random problem configurations with 1% probability of containing a flawed variable

n	m	p_1	p_2
50	10	1	0.65
50	10	0.23	0.75
50	10	0.04	0.87
50	50	1	0.93
50	50	0.75	0.94
50	50	0.04	0.99

We would expect that randomly generated problems with $p_2 \geq 0.5$ would tend to be insoluble as n increases, using both flawed and flawless constraints, as the majority of domain pairs are disallowed. However, we are interested in determining if using flawed constraints will produce a large number of insoluble problems in our dataset. In Table 3.1, we present several of our configurations with a 1% probability of containing a flawed variable. For each of the given configurations, increasing n,p_1, or p_2, or decreasing m, will increase the probability of a variable being flawed. From this we can see that for our problems flawed variables are unlikely to occur until well above $p_2 = 0.5$, the point at which flawless variables tend to be insoluble, as $n \to \infty$. Therefore, the use of flawed variables should not have a significant impact on our work.

We have three criteria for a problem instance to be useful in our experimentation: first, there must be a difference in solving time that is ≥ 1 millisecond; second, the problem must be completed by both solvers; and third, the problem must be connected. Disconnected problems are not of interest as each connected component can be treated as a separate problem. We are only interested in problems that did not reach the timeout, as we need to know the relative time to solve the problem between both solvers. Finally, only problems where there was a measurable difference in solution time are of interest, as there is no value in predicting between solvers when there is not gain to be made. Thus, there were a total of 4908 remaining problems, which we refer to as the Random Problems Dataset.

In order to test the reproducibility of the runtimes measured, we tested a group of problems to measure the distribution of their runtimes. We selected random problems with runtimes between 1 and 3 minutes. Each problem was then solved 10 times in order to collect a distribution of runtimes for solving each problem. In table 3.2 we present the average mean solving time, the average standard deviation, and the average coefficient of variation. Additionally, we have included the mean, standard deviation, and coefficient of variation for the three problem instances with the highest mean, the highest standard deviation, and the highest coefficient of variation. We found that the distribution of runtimes was relatively consistent, and thus we assume that the runtimes for the other problems we consider are relatively consistent as well. This assumption allows us to explore a larger dataset, as we do not need to replicate each data point several times to control for noise caused by the operating system and background processes.

Table 3.2: Runtime distributions of random problems with solving time between 1 and 3 minutes. Each problem was solved 10 times. The average runtime, standard deviation of runtime and coefficient of variation of runtime over the data set are labeled *average*. The three problems with the largest mean, largest standard deviation and largest coefficient of variation are presented as well.

	μ	σ	C
average	81460	1599	0.02
highest μ	164951	1724	0.01
highest σ	96534	7062	0.07
highest C	84315	6462	0.08

As we can see from Figures 3.1 and 3.2, most of the random problems we examined are solved faster with FC than with AC. Figures 3.1 and 3.2 present exactly the same data, but have linear and logarithmic scales on the axes, respectively. Of particular interest, the hardest problems, or those with the longest runtime, are all solved faster with FC than with AC, which can be seen in Figure 3.1. This is due to the sampling methodology we used. Other works comparing AC and FC have focused on small sections of the problem spaces or examined along an axis, such as varying p_1 and p_2, while maintaining a fixed τ (Sabin & Freuder, 1994; Bessière & Régin, 1996). The uniform sampling results in exploring large regions where FC tends to outperform AC, but these regions are missed in other sampling schemes. Often, a fixed τ value is used to explore only the hardest regions of the problem space. However, we do not know that an arbitrary problem will be a hard problem, so we did not choose to bias our data collection in favour of hard problems. We can see

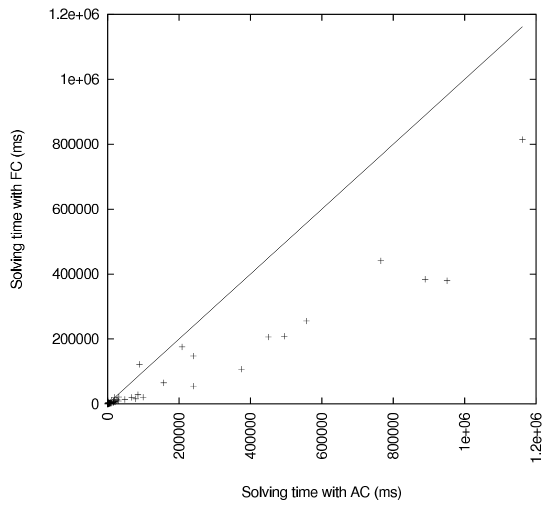


Figure 3.1: AC vs FC solving time: Random Problems Dataset

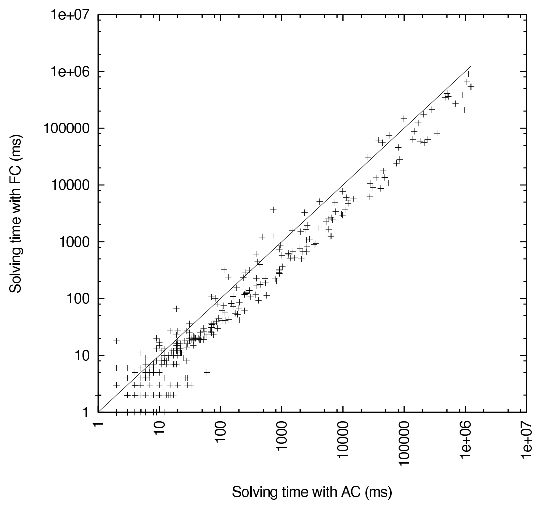


Figure 3.2: AC vs FC solving time: Random Problems Dataset, log-log plot

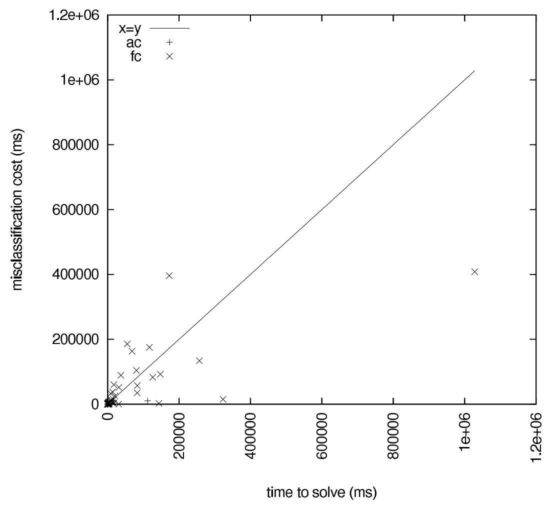


Figure 3.3: solving time vs misclassification cost: Random Problems Dataset

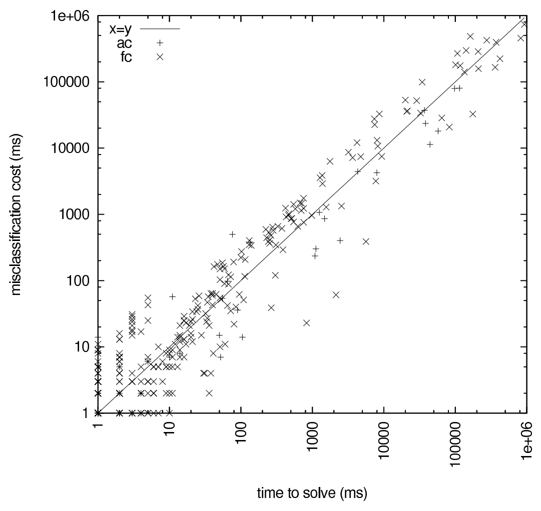


Figure 3.4: solving time vs misclassification cost: Random Problems Dataset, log-log plot

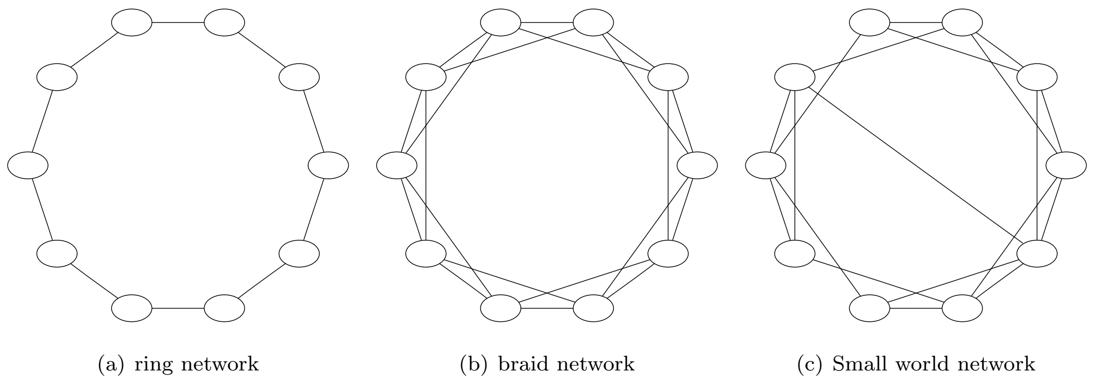

(a) ring network (b) braid network (c) Small world network

Figure 3.5: Small worlds are generated by forming a ring, as in (a), then adding edges to neighbours-of-neighbours, as in (b), and finally randomly reassigning some edges, as in (c).

from Figures 3.3 and 3.4, the differences between easy and large problems are several orders of magnitude. From 3.4 we can observe that the misclassification costs are roughly in line with the solving time, and thus mistakes in choosing a propagator are significant, as expected.

3.5.3 Small World Problems

Random problems are useful in that they can be generated very quickly, making them convenient for experimentation. However, they are unlike "real" problems because they exhibit no patterns or local structure. In order to test our methods on problems that are structured, we make use of small world problems. The small world problems we investigate cover the range of networks described by Watts and Strogatz (1998) as "small worlds" as well as some problems that are both too regular and too irregular in their structure to be small worlds. We used random constraints on each edge of the graph. Small world graphs can be generated by first forming a ring network (Fig. 3.5(a)), adding edges between each node and its immediate neighbours, and then adding edges to the next node beyond its neighbour in the ring. The "braid width" corresponds to the number of neighbours in each direction that a node is connected to. After a braid is formed (Fig. 3.5(b)) a small fraction of the edges are randomly "rewired" to new nodes in the graph, resulting in irregularities in the previously very regular structure, to create the shortcuts within the small world graph (Fig. 3.5(c)). Small world graphs were generated with the following configurations: n=[10,30,50], m=[10,30,50], braid width=[2,3,4,5,6], fraction of edges rewired =[0.00..0.2] in increments of 0.01, p_2 = [0.01..1] in increments of 0.01. There were a total of 90,000 small world problems. From the 90,000 problem instances generated, 46,503 usable data points were collected after removing problems that either were disconnected, had no difference in their solving times between AC and FC, or timed out. These remaining problems we denote as the Small World Problems Dataset.

Figures 3.6 and 3.7 indicate a more even split between problems solved faster with AC and FC than was observed in the Random Problems Dataset. Figures 3.6 and 3.7 present exactly the same

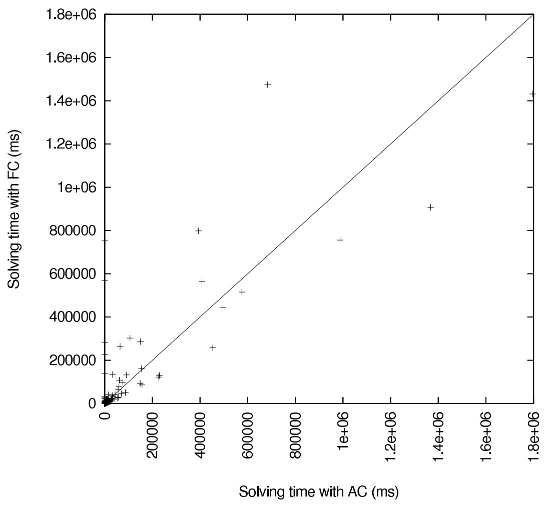


Figure 3.6: AC vs FC solving time: Small World Problems Dataset

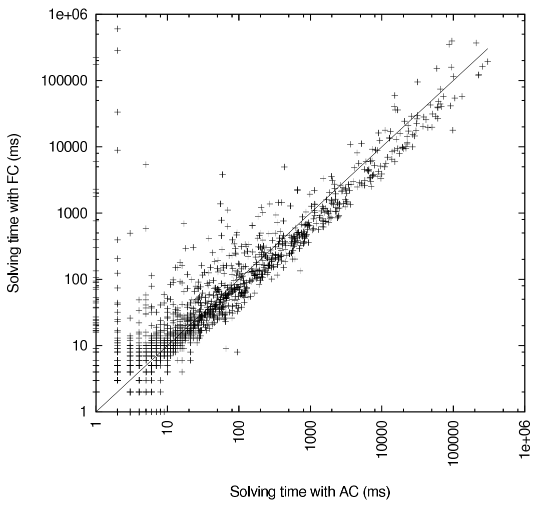


Figure 3.7: AC vs FC solving time: Small World Problems Dataset, log-log plot

data, but have linear and logarithmic scales on the axes, respectively. Due to the parametrization of the small world problems, there is a greater proportion of problems with small p_1, which previous works on random problems have found to be a region where AC tends to have stronger performance than FC. In Figure 3.7 we can see that some problems solved extremely quickly with AC are solved extremely slowly with FC (instances appearing in the top left corner).

3.5.4 Quasigroup With Holes Problems

Quasigroups with holes (QWH), as described in Section 2.1.2, are formed by taking complete quasigroups (Fig. 3.10(a)) and "poking holes" or deleting values in the problem (Fig. 3.10(b)). By starting with a completed quasigroup and removing values, we can guarantee that each problem has a solution. QWH have two control parameters, n, which controls the side length in the square (n is also the number of values each variable can take), and the fraction of holes, which determines how many of the cells are marked as empty. We generated quasigroups with holes with $n = 10$ to $n = 15$ and the fraction of holes between 0.25 and 0.5 in 1 hole increments. Disequality constraints are imposed between all cells of the quasigroup in each row, as well as each column. Gomes and Shmoys (2002) used $n = 11$ to $n = 15$ with the fraction of holes between 0.25 and 0.5 and that found the hardest problems occurred around 0.3 to 0.35. We considered a total of 2440 QWH problems formed by using 6 values for n, and varying the fraction of holes from 0.25 to 0.5, in 1 hole increments (26 problems with $n = 10$, 31 with $n = 11$, 37 with $n = 12$, 43 with $n = 13$, 50 with $n = 14$, 57 with $n = 15$), totalling 244 problems, with 10 instances of each parametrization, totalling 2440. 2340 usable data points were collected from these experiments, which we will call the QWH Problems Dataset. A much larger fraction of the QWH problems were usable than the random and small world problems. This is because the quasigroup problems were significantly larger than the

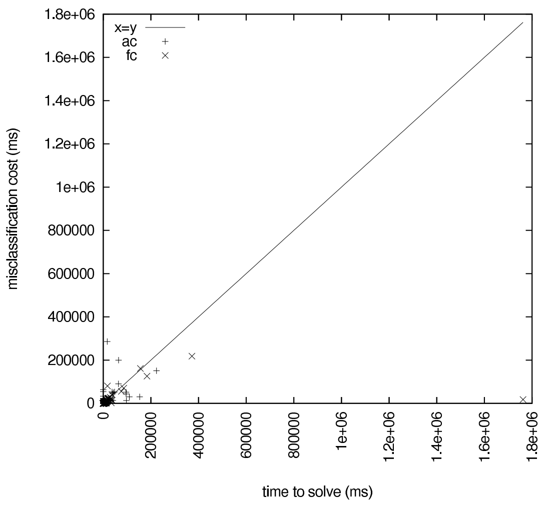


Figure 3.8: Solving time vs misclassification cost: Small World Problems Dataset

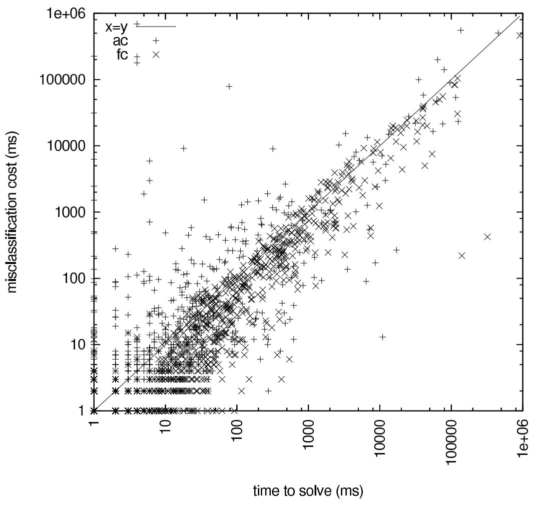
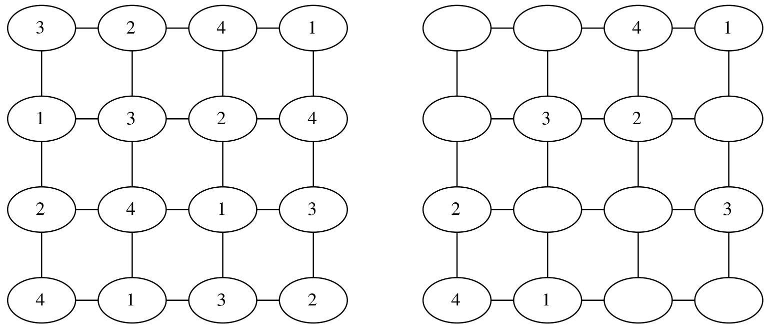


Figure 3.9: Solving time vs misclassification cost: Small World Problems Dataset, log-log plot



(a) An example complete quasigroup.

(b) An example of a quasigroup with holes.

Figure 3.10: Quasigroups with holes (QWH) are generated by forming a quasigroup, as in (a), then un-assigning values, as in (b).

other problem classes; thus, fewer problems were solved in an immeasurably small amount of time.

From Figure 3.11, we can see that the instances that take the longest to solve take much longer to solver with FC than with AC. Figure 3.12 shows that generally problems that are solved quickly are solved with FC and problems that are slower to solve are solved with AC. Figures 3.11 and 3.12 present exactly the same data, but have linear and logarithmic scales on the axes, respectively. Figures 3.13 and 3.14 show that the misclassification costs of quickly solved problems tend to be small (data points below the x=y line), but the costs of applying the wrong solver are much larger for problems that take longer to solve. In fact, for the slowest problems to solve, the difference in solving time can be an order of magnitude.

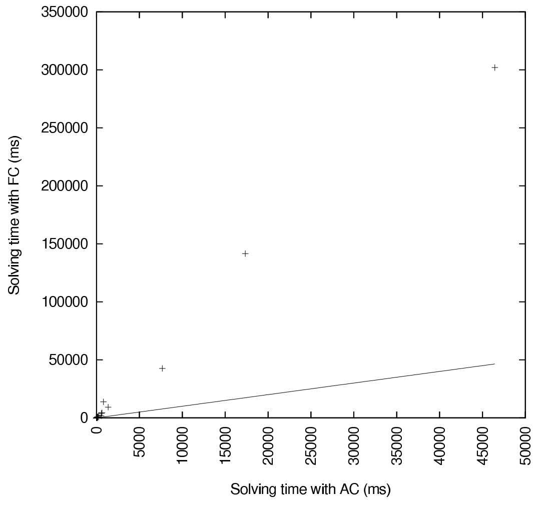


Figure 3.11: AC vs FC solving time: QWH Problems Dataset

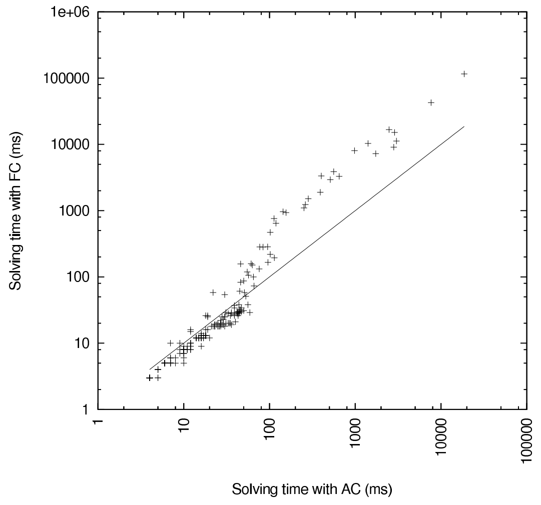


Figure 3.12: AC vs FC solving time: QWH Problems Dataset, log-log plot

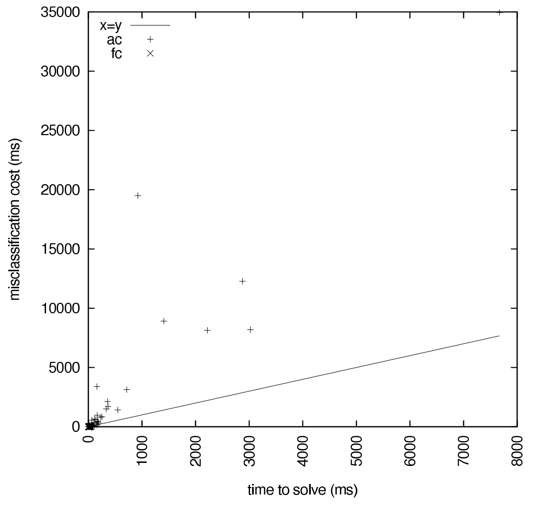


Figure 3.13: Solving time vs misclassification cost: QWH Problems Dataset

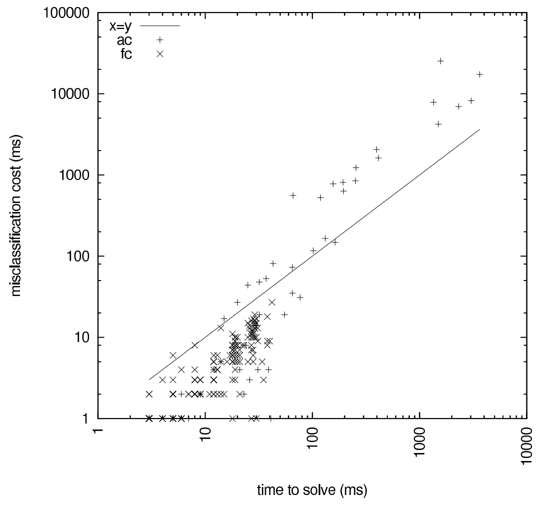


Figure 3.14: Solving time vs misclassification cost: QWH Problems Dataset, log-log plot

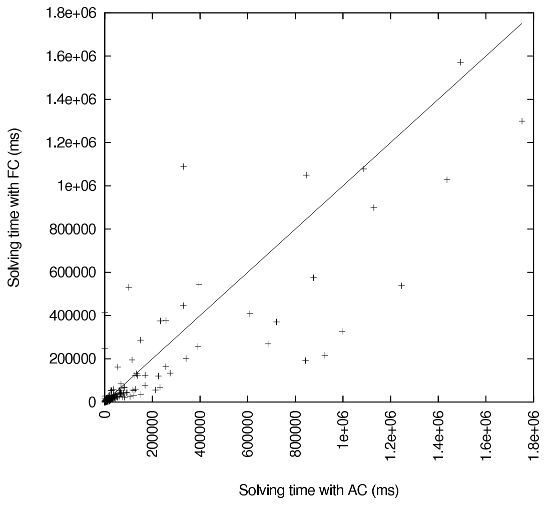


Figure 3.15: AC vs FC solving time: All Problems Dataset

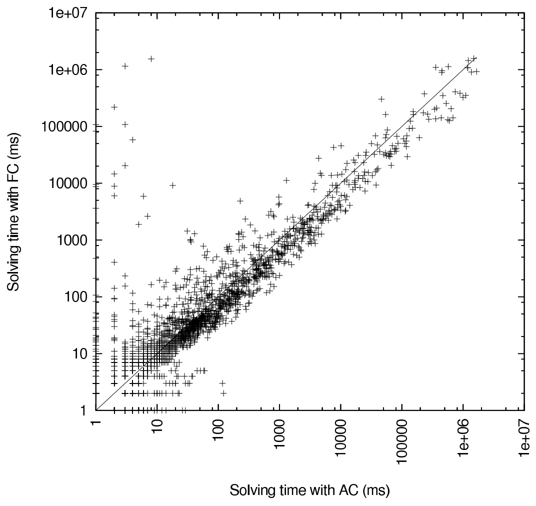


Figure 3.16: AC vs FC solving time: All Problems Dataset, log-log plot

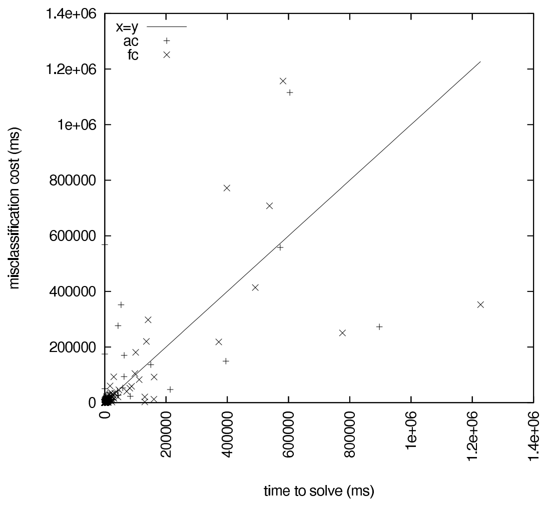


Figure 3.17: Solving time vs misclassification cost: All Problems Dataset

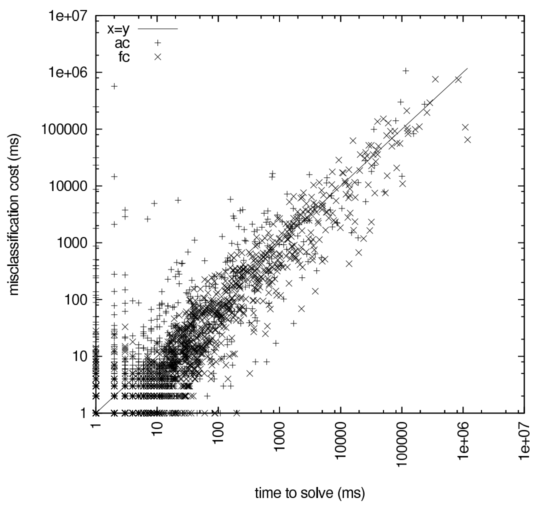


Figure 3.18: Solving time vs misclassification cost: All Problems Dataset, log-log plot

3.5.5 Combined Problem Set

Finally, we have created an "All Problems" dataset by combining all the datapoints from the Random, Small Worlds, and QWH problem sets. This dataset will enable us to train a classifier on a heterogeneous problem class, which may contain features not found in the classifiers build on homogeneous problem sets. In Figures 3.15 through 3.18, we can see the solving times for all problems summed together. However, because there are many more small world problems than random or QWH problems, the small worlds dominate the figures. Figures 3.15 and 3.16 present exactly the same data, but have linear and logarithmic scales on the axes, respectively.

We chose to combine all the data points, rather than choose a subset of the data points such as an even distribution of each problem class. It is not possible for us to predict the distribution of problems that would be encountered in a real world application, so we have chosen to train our classifiers using all of the available data. Assessing the impact of the problem class distribution on the learning task is left to future work.

3.5.6 Summary

In this chapter, we presented our metareasoning CSP solver design, including AC and FC algorithms, and a discussion of the learning algorithms used. We have also presented our CSP datasets and their AC and FC runtime distributions. Additionally, we presented the problem attributes we have collected and the data cleansing methods we used to remove problems from our datasets.

Algorithm 1 SEARCH (adapted from Dechter (2003), Figure 5.7)

```
Input: A constraint network R = (X,D,C)
Output: Either a solution, or notification that the network is inconsistent
  $i \leftarrow 1$
  $K \leftarrow X$
  while $1 \le i \le n$ do
    $x_i \leftarrow$ SELECT-VARIABLE($X$)
    $a \leftarrow$ SELECT-VALUE($x_i$)
    if $a = null$ then
      $i \leftarrow i - 1$
      reset each $D'_k, k > i$, to its value before $x_i$ was last instantiated
      $K \leftarrow K \cup \{x_i\}$
    else
      remove all domain values other than $a$ from $D_i$
      $K \leftarrow K \setminus \{x_i\}$
      if PROPAGATE($x_i$,$K$) is true then
        $i \leftarrow i + 1$
      else
        reset each $D'_k, k > i$, to its value before $x_i$ was last instantiated
        remove $a$ from $D_i$
        $K \leftarrow K \cup \{x_i\}$
      end if
    end if
  end while
  if $i = 0$ then
    return "inconsistent"
  else
    return instantiated values of $\{x_1, \ldots, x_n\}$
  end if
```

Algorithm 2 PROPAGATE-AC-3 (adapted from Mackworth (1977a))

Input: an assignment x_{cur}, a list of free variables K

Output: True if the propagation does not find an inconsistency, false otherwise

```
$Q$={}
for all free variables, $x_k$ in $K$ do
  if $C_{cur,k} \in C$ then
    Q.push($(cur, k)$)
  end if
end for
while !Q.empty do
  revised = false
  $(x_i, x_k)$ = Q.pop()
  if REVISE($x_i$,$x_k$) = true then
    for all free variables, $x_j$ in $K$ do
      if $C_{k,j} \in C$ then
        Q.push($k, j$)
      end if
    end for
  end if
  if $D_k$ is empty then
    return false
  end if
end while
return true
```

Algorithm 3 PROPAGATE-FC

Input: the assigned variable, x_i, a list of free variables K

Output: True if the propagation does not find an inconsistency, false otherwise

```
for all free variables, $x_k$ in $K$ do
  if $C_{i,k} \in C$ then
    REVISE($x_i$,$x_k$)
  end if
  if $D_k$ is empty then
    return false
  end if
end for
return true
```

Algorithm 4 REVISE

Input: two variables, x_i,x_j

Output: true if a value is removed from D_j. As a side effect, all remaining values in D_j are supported by values in D_i

```
revised = false
for all values $a$ in $D_j$ do
    if there is no $b$ in $D_i$ such that CONSISTENT($x_i = b$, $x_j = a$) then
        remove $a$ from $D_j$
        revised = true
    end if
end for
return revised
```

Chapter 4

Experimental Results

In this chapter, we present our experiments and findings. In Section 4.1 we present the information gain of the CSP attributes we examined as well as a method for finding important subsets of the attributes. Section 4.2 examines the accuracy with which we can make correct predictions about which solver to apply. In Section 4.3 we take a cost sensitive approach to answering the same questions as Section 4.1 and 4.2. We consider the task of classifying problems of a new, previously unseen type in Section 4.4. Finally, in Section 4.5 we present our findings regarding the practicality of metareasoning for solving CSPs.

It is difficult to compare our results to other published works, even ones using AC and FC, because there are so many implementation details that affect the number of constraint checks or backtracks. Without full knowledge of all of the implementation choices, we may find different results when solving exactly the same problems. As an example, Sabin and Freuder (1994) used AC-4 and show it is superior to FC over the almost the entire random problem space, whereas we show FC superior to AC-3 for many problems. Additionally, Bessière and Régin (1996) use AC-7 for their work comparing AC and FC and found AC to be better than FC for almost all problems considered. Clearly, the specifics of the Arc Consistency implementation can lead to differing results.

4.1 Information Value of CSP Attributes

In this section, we investigate which CSP attributes are valuable in classifying problem instances according to the preferred solving method.

In order to make predictions about which solving algorithm should be applied to a given CSP, we must have an effective set of attributes with which to measure or describe a CSP. Not all measures of a CSP are equal in their ability to discriminate between a choice of solving techniques. We have collected 24 attributes of each CSP, as described in Section 3.5.1.

To evaluate the merit of the problem attributes, we have ranked them based on their information gain. The information gain of an attribute is the increase in the likelihood of an outcome, given knowledge of the attribute. In the case of predicting which solving algorithm to use, we have a baseline distribution over all problem instances. Given knowledge of one attribute, the probability distribution of which algorithm is best suited to the problem will change. Knowledge of some attributes will greatly increase the likelihood of one or another outcome, whereas knowledge of other attributes may provide little or no information. The attributes are ranked by information gain in Table 4.1. The formula for determining the information gain of an attribute is equal to the information value of the entire data set, minus the information value of the data after splitting on the attribute (Shannon & Weaver, 1948).

One of the issues faced when determining the information gain of a numeric attribute is determining how to discretize the attribute. If each recorded value is treated individually, we have a perfect mapping from the numeric attribute to the classification. As an example, if we treat social insurance numbers of people as an attribute and gender as the classification we wish to make, then each SIN uniquely identifies an individual and thus can predict gender with 100% accuracy. Unfortunately, this has no predictive power for new, unseen SINs. However, if we discretize SINs into categories of above and below the mean, then we would expect SIN to have very low information gain for predicting gender. Therefore, we want to choose a discretization somewhere between these two extremes. In truth, we would expect the SIN to have very low information gain when predicting gender.

In our experimentation, we have determined information gain values after discretizing numeric attributes using the minimum description length principle (MDL) (Fayyad & Irani, 1993). MDL aims to minimize the amount of information required to describe the data. Given a set of attributes and a classification for a dataset, each data point can be encoded as a $\langle$attribute,classification$\rangle$ pair. However, this requires $f(n)$ space, where $f(n)$ is $O(n)$, to store the classifications. Instead, we can assume that all the data points are of one classification (base class) and then specify the exceptions. This reduces the space to store the classifications for a two class dataset to $\frac{f(n)}{2} + 1$. Furthermore, the dataset can be partitioned into regions with different base classes. MDL will make these partitions whenever the total amount of information required to specify the classifications of the data is reduced.

The most useful attributes for making predictions will depend on the learning algorithm, as some algorithms are better able to handle noisy or redundant data. Because the information gain of each attribute is assessed individually, redundant attributes will be given the same rank. The information gain ranking algorithm pays no attention to redundant information, so identical attributes with different names will be ranked equally high. In practice, duplicate attributes are of no real use. Therefore, simply selecting the top 5 attributes in the ranking may not produce optimal results. Counter to intuition, having more attributes does not always lead to better predictors. For example, when using the k-nearest neighbours, all attributes are typically given equal weight, so there is no capacity for the algorithm to discriminate the few most useful attributes from the many non-useful ones.

We have presented the attribute rankings for the All Problems Dataset as well as the subsets by problem type in Table 4.1, as we expected that the most relevant attributes would change depending on what part of the problem space we examine (e.g. small worlds vs random, "quick" problems vs long problems, and high misclassification cost vs low misclassification cost).

In Table 4.1 we present the problem attributes recorded and for each dataset, their rank by information gain, and the actual information gain. From the table we can see that measures of tightness (i.e. tightest constraint, loosest constraint, tightness variance, p_2) and τ are most relevant to the entire problem set, and to the Small World Problems Dataset, the latter of which dominate the All Problems Dataset. The measure of random problem hardness, τ, is the only measure which is ranked highly in all data sets. Furthermore, note that many of the attributes are tied as the fourth most informative attribute for QWH problems. This is because all of these attributes are totally dependent on problem size in QWH problems; thus, they provide the same amount of information as knowing the number of variables in the problem.

We also used an attribute selection scheme, Correlation-based Feature Selection (CFS) (Hall, 2000). CFS aims to find subsets of attributes with high correlation between each attribute and the classification, but low correlation between each attribute. The possible attribute subset space is searched best first, terminating after 5 consecutive subset expansions fail to produce improved subsets. The most useful attribute subsets found by the CFS algorithm do not correspond to the top n attributes from the ranking. This is because there is redundant information among these attributes. Furthermore, attributes may not be perfectly redundant, but instead may contain significant overlap in their information. From the attribute subsets in Table 4.2, we can see that τ is selected to be in all the attribute subsets, as is the average domain size.

Interestingly, some attributes, such as *domain size variance*, *width*, and *acyclic* are selected in the all data set, but not in the other datasets. This indicates that they may not be useful within a problem class, but can be used to distinguish between problem classes. For example, only QWH problems will have variance in their domain sizes (as the domain size for random and small world problems is fixed), but it must not be a useful parameter within the QWH Problems Dataset or it would have been selected by the attribute subset. From the attribute rankings, we can see that domain size variance is the top ranked attribute for the QWH Problems Dataset, but it was not selected for the attribute subset. Therefore, the information provided by the domain size variance must not be as useful as the information provided by τ and average domain size, which were the attributes selected to be in the subset.

We can conclude that some attributes are useful for all the problem types we examined, such as τ and average domain size. Some other attributes are valuable within a dataset, such as the average degree and clustering coefficient 2 within the Random Problems Dataset. Finally, some

Table 4.1: Attributes ranked by information gain for each of 4 datasets. The actual information gain value is given in parentheses.

attribute	All		Random		Small World		QWH	
tightest constraint	1	(0.1357)	20	(0.0147)	1	(0.1613)	4	(0.0338)
loosest constraint	2	(0.1202)	21	(0.0144)	2	(0.1453)	4	(0.0338)
τ	3	(0.0971)	1	(0.098)	3	(0.1082)	2	(0.3625)
tightness variance	4	(0.0865)	19	(0.0236)	4	(0.1045)	20	(0)
p_2	5	(0.0753)	22	(0.0116)	5	(0.0887)	4	(0.0338)
domain size average	6	(0.0681)	14	(0.0265)	6	(0.0588)	3	(0.348)
domain size smallest	7	(0.0531)	14	(0.0265)	6	(0.0588)	20	(0)
domain size largest	8	(0.0517)	14	(0.0265)	6	(0.0588)	4	(0.0338)
p_1	9	(0.0402)	9	(0.0696)	9	(0.0231)	4	(0.0338)
average degree	10	(0.0393)	3	(0.0865)	14	(0.0177)	4	(0.0338)
number of edges	11	(0.0386)	6	(0.0788)	13	(0.0181)	4	(0.0338)
width	12	(0.0385)	2	(0.0894)	11	(0.0186)	4	(0.0338)
min degree	13	(0.0367)	4	(0.0852)	12	(0.0182)	4	(0.0338)
average path length	14	(0.0360)	8	(0.0719)	10	(0.0194)	4	(0.0338)
path length variance	15	(0.0351)	7	(0.0719)	15	(0.0175)	4	(0.0338)
max degree	16	(0.0323)	5	(0.0837)	17	(0.0122)	4	(0.0338)
longest path	17	(0.0232)	13	(0.0473)	16	(0.0148)	20	(0)
induced width	17	(0.0232)	10	(0.0670)	20	(0.0076)	4	(0.0338)
clustering coefficient 1	19	(0.0207)	12	(0.0500)	18	(0.0119)	4	(0.0338)
clustering coefficient 2	20	(0.0190)	11	(0.0504)	19	(0.0118)	4	(0.0338)
domain size variance	21	(0.0189)	24	(0)	23	(0)	1	(0.3782)
degree variance	22	(0.0154)	17	(0.0260)	22	(0.0025)	20	(0)
variables	23	(0.0063)	18	(0.0246)	21	(0.0037)	4	(0.0338)
acyclic	24	(0.0001)	23	(0.0034)	23	(0)	20	(0)

Table 4.2: Attributes selected by the CFS algorithm are marked with X for each dataset.

attribute	All	Random	Small World	QWH
tightest constraint	X		X	
loosest constraint	X			
τ	X	X	X	X
tightness variance	X		X	
p_2				
domain size average	X	X	X	X
domain size smallest				
domain size largest				
p_1				
average degree		X		
number of edges				
width	X			
min degree				
average path length				
path length variance				
max degree				
longest path				
induced width		X		
clustering coefficient 1				
clustering coefficient 2		X		
domain size variance	X			
degree variance				
variables		X	X	
acyclic				

Table 4.3: Summary of j48 prediction accuracy over 4 datasets.

	Accuracy (%)	κ	AC/FC distribution
All	80.23	0.5334	17854/35897
Random	88.18	0.5336	769/4139
Small World	78.85	0.5218	16594/29909
QWH	91.45	0.7245	491/1849

Table 4.4: j48 performance on the All Problems Dataset. The column marked j48 indicates performance using all of the 24 attributes, whereas the second and third columns use the Expert and CFS subsets. Each column contains prediction accuracy, followed by κ.

runtime difference	j48		Expert		CFS	
$\geq$ 1 millisecond	80.23	(0.5334)	80.90	(0.5497)	79.76	(0.5285)
$\geq$ 10 milliseconds	93.77	(0.8672)	93.59	(0.8635)	91.71	(0.8239)
$\geq$ 100 milliseconds	96.37	(0.9214)	96.03	(0.914)	93.75	(0.8644)
$\geq$ 1 second	97.20	(0.9381)	96.49	(0.9221)	95.24	(0.8946)

attributes may be useful for distinguishing between problem sets, such as domain size variance, or width.

4.2 Solving Method Prediction Accuracy

In this section, we investigate the prediction accuracy of j48 decision trees over our entire dataset as well as subsets for each problem type.

We present our results in terms of accuracy and κ. Accuracy reflects the percentage of classifications made correctly, but can be biased by the distribution of the data. For example, if the dataset is 90% class A and 10% class B, then a classifier that classifies every instance to be of class A has 90% accuracy. Taken alone, 90% accuracy seems quite good, that is until the underlying data distribution is known. κ represents the agreement between the classifier and the data, corrected for chance agreement. κ values range from -1 to 1, and the previously mentioned classifier would have a κ value of 0 (Cohen, 1960; Witten & Frank, 2005). The κ values are calculated by $\kappa = \frac{P-P_e}{1-P_e}$, where P is the fraction of instances correctly classified and P_e is the expected chance agreement (Witten & Frank, 2005).

Table 4.3 provides a summary of the j48 accuracy, κ, and the underlying distribution of problem instances. We can see that the standard j48 algorithm, using all problem attributes, produces 78% to 91% accuracy on the datasets, with κ between 0.52 and 0.72. These κ values indicate that the

Table 4.5: j48 performance on the Random Problems Dataset. The column marked j48 indicates performance using all of the 24 attributes, whereas the second and third columns use the Expert and CFS subsets. Each column contains prediction accuracy, followed by κ.

runtime difference	j48		Expert		CFS	
$\geq$ 1 millisecond	88.18	(0.5336)	88.90	(0.5347)	88.69	(0.5333)
$\geq$ 10 milliseconds	96.53	(0.8157)	97.07	(0.8479)	97.56	(0.8718)
$\geq$ 100 milliseconds	96.74	(0.8365)	96.56	(0.8249)	97.01	(0.8474)
$\geq$ 1 second	98.13	(0.9073)	98.29	(0.9145)	97.98	(0.9002)

Table 4.6: j48 performance on the Small World Problems Dataset. The column marked j48 indicates performance using all of the 24 attributes, whereas the second and third columns use the Expert and CFS subsets. Each column contains prediction accuracy, followed by κ.

runtime difference	j48		Expert		CFS	
$\geq$ 1 millisecond	78.85	(0.5218)	79.34	(0.5302)	79.44	(0.5326)
$\geq$ 10 milliseconds	92.78	(0.8523)	92.66	(0.8499)	92.30	(0.8425)
$\geq$ 100 milliseconds	95.37	(0.903)	95.37	(0.903)	95.23	(0.9002)
$\geq$ 1 second	96.59	(0.9287)	97.32	(0.9441)	96.85	(0.9343)

decision tree correctly classifies between 50% and 70% of the additional instances not accounted for by chance agreement.

In Tables 4.4-4.7, we examine the problem sets individually, splitting the datasets according to the relative importance of instances. As mentioned earlier, we discarded data where there was no measurable difference in runtime between AC and FC, as we have no preference for which solver is used when there is no difference in runtime. The data points included in each row in the tables are selected by choosing only the instances that meet the indicated difference in runtime. Thus, the bottom row of each table corresponds to the accuracy of decision trees trained and tested on only the instances where there was a runtime difference of 1 second or more. It should be noted that as the difference in runtime criteria increases, the number of instances that satisfy the criteria decreases. For the chosen subset, we then perform 10-fold cross validation, and present the results as the accuracy percentage and κ value.

For each subset of the data (by importance), we evaluated three j48 trees: one using all the attributes, one using only $\langle n, m, p_1, p_2, \tau \rangle$, and one using the subset selected by CFS (refer to Table 4.2). Also, note that the rows in the tables are cumulative; that is, any sample problem where the difference in runtime is $\geq$ 1 second will appear in all four rows. The last two columns correspond two decision trees built using two subsets of the attributes for training. The first used

Table 4.7: j48 performance on the QWH Problems Dataset. The column marked j48 indicates performance using all of the 24 attributes, whereas the second and third columns use the Expert and CFS subsets. Each column contains prediction accuracy, followed by κ.

runtime difference	j48		Expert		CFS	
$\geq$ 1 millisecond	91.45	(0.7245)	91.50	(0.7235)	90.56	(0.7062)
$\geq$ 10 milliseconds	93.93	(0.8783)	94.23	(0.8842)	91.57	(0.8305)
$\geq$ 100 milliseconds	100	(0)	100	(0)	100	(0)
$\geq$ 1 second	100	(0)	100	(0)	100	(0)

the attributes, $\langle n, m, p_1, p_2, \tau \rangle$. The first four of these attributes are commonly used to describe random problems, and τ is frequently cited as a measure of instance difficulty for random problems (Gent et al., 1995, 1996). This attribute subset represents those that we believed would be most useful in distinguishing between problem instances. Thus, it will be referred to as the "Expert" subset.

There are two reasons for investigating the use of several attribute sets. First, we would like to know if the attributes commonly used for describing CSPs are adequate for differentiating problems by how they should be solved. Second, we would like to know how the performance changes when we use all attributes versus a subset of attributes. This is of interest because it is much faster to collect only a subset of attributes, rather than the full set.

As intuition would suggest, the problems for which there is the least difference in runtime are also the problems where it is most difficult to distinguish which solver should be applied. Furthermore, as the difference in runtime increases, the number of sample problems decreases. We can see that in all tables, increasing the threshold of runtime difference results in an increase in prediction accuracy. Furthermore, it is more likely that there will be data points with small differences in runtime caused by random noise. However, it is unlikely that there will be data points with very large differences in runtime due to noise.

From Tables 4.4-4.7 we can see that attribute subset selection makes a small difference in the accuracy and κ value of the classifier. Furthermore, the Expert subset tends to slightly outperform the CFS subset.

In the last two rows of Table 4.7 it appears that classification accuracy is 100%. However, the dataset only contains instances solved faster with AC than with FC. Therefore, although the accuracy is 100%, the κ value is 0, as there is no increase in prediction accuracy over the agreement expected due to chance.

From this section we are able to conclude that the task of predicting which solver to apply becomes easier as the differences in runtime in the dataset become larger. Furthermore, we can see that decision trees trained using the Expert and CFS attribute subsets perform on par with or slightly better than decision trees trained from all attributes.

4.3 Relative Importance of Individual Problem Instances

In the previous sections we identified subsets of the attributes that are useful for making classifications. In this section, we aim to improve the performance of the classifier. There is large variance in the importance of correct predictions, depending on the problem instance. On some problems the difference in solving time between algorithms is immeasurably small. However, on other problems the difference can be several orders of magnitude. The standard j48 implementation treats all training instances as equally important.

In order to improve overall performance, we can train our learning algorithm to minimize the average misclassification cost, where the cost of a misclassification is the difference in solving time between the two algorithms. This differs from the earlier approach in that not all instances are equally weighted. We weight each instance by its misclassification cost, such that an instance where the runtime difference between the two algorithms is 10ms is now 10 times as important as an instance with a runtime difference of 1ms. To achieve this, we re-sampled the dataset, multiplying the number of occurrences of each instance by its misclassification cost. Then, we determined the average misclassification cost using the basic solver and the cost sensitive solver.

As we have done earlier, we have used attribute selection algorithms to rank the information gain of each attribute and to select the attribute subset of highest information value. Table 4.8 shows the attribute rankings, and the information gain of each attribute in parentheses. In the cost sensitive context, we can see see that τ is highly ranked in all four datasets, p_2 and both clustering coefficients are ranked highly in the All, Random, and Small World datasets, while average domain size and domain size variance are ranked highly in the QWH dataset. As was seen previously in table 4.1, again we see that many of the attributes for QWH problems are determined by the number of variables in the problem, and thus provide the same information gain as the number of variables.

Table 4.9 shows the attribute subsets chosen by the CFS attribute selection algorithm. As noted from Table 4.8, τ is a useful attribute and is chosen by CFS for all 4 attribute subsets. p_2 and average domain size are both chosen for three of the four subsets. Interestingly, the average domain size was only ranked highly in the QWH dataset. This confirms again that the best subset of the attributes may not be the top few attributes in the information gain ranking.

In Table 4.10 we present the total misclassification cost of several decision trees on our four data sets. Total misclassification cost represents the sum of costs associated with all the misclassifications the decision tree makes on the specified data set.

Table 4.8: Attributes ranked by information gain for each of 4 cost sensitive datasets. The actual information gain value is given in parentheses.

attribute	All		Random		Small World		QWH	
tightest constraint	8	(0.8281)	7	(0.2618)	7	(0.7844)	4	(0.0051)
loosest constraint	6	(0.8567)	5	(0.2664)	8	(0.7747)	4	(0.0051)
τ	1	(0.9629)	1	(0.2880)	1	(0.8898)	3	(0.0224)
tightness variance	12	(0.5961)	8	(0.2234)	10	(0.4431)	20	(0)
p_2	3	(0.9600)	2	(0.2878)	2	(0.8875)	4	(0.0051)
domain size average	20	(0.1500)	19	(0.0529)	19	(0.1704)	1	(0.0225)
domain size smallest	22	(0.1491)	19	(0.0529)	19	(0.1704)	20	(0)
domain size largest	21	(0.1493)	19	(0.0529)	19	(0.1704)	4	(0.0051)
p_1	9	(0.6803)	16	(0.1409)	9	(0.4455)	4	(0.0051)
average degree	11	(0.6517)	14	(0.1446)	14	(0.3808)	4	(0.0051)
number of edges	10	(0.6610)	14	(0.1446)	11	(0.4005)	4	(0.0051)
width	14	(0.5825)	11	(0.1538)	13	(0.3814)	4	(0.0051)
min degree	16	(0.5733)	10	(0.1565)	12	(0.3982)	4	(0.0051)
average path length	7	(0.8458)	12	(0.1505)	6	(0.7875)	4	(0.0051)
path length variance	5	(0.8812)	13	(0.1481)	5	(0.8636)	4	(0.0051)
max degree	15	(0.5769)	9	(0.1640)	16	(0.3182)	4	(0.0051)
longest path	18	(0.4188)	18	(0.1045)	17	(0.2368)	20	(0)
induced width	17	(0.5030)	17	(0.1243)	18	(0.1985)	4	(0.0051)
clustering coefficient 1	4	(0.9453)	3	(0.2876)	4	(0.8786)	4	(0.0051)
clustering coefficient 2	2	(0.9604)	4	(0.2870)	3	(0.8871)	4	(0.0051)
domain size variance	23	(0.0525)	24	(0)	23	(0)	1	(0.0225)
degree variance	13	(0.5870)	6	(0.2643)	15	(0.3511)	20	(0)
variables	19	(0.1590)	22	(0.0247)	22	(0.0968)	4	(0.0051)
acyclic	24	(0.0001)	23	(0.0004)	23	(0)	20	(0)

Table 4.9: Cost sensitive CFS attribute subsets. Attributes marked with X are chosen by the CFS algorithm.

attribute	All	Random	Small World	QWH
tightest constraint				
loosest constraint				
tau	X	X	X	X
tightness variance				
p_2	X	X	X	
domain size average	X		X	X
domain size smallest				
domain size largest				
p_1				
average degree				
number of edges				
width				
min degree				
average path length				
path length variance				
max degree				
longest path				
induced width				
clustering coefficient 1				
clustering coefficient 2		X		
domain size variance				X
degree variance				
variables				
acyclic		X		

First, we consider the misclassification cost of the decision tree created without cost information (see column labelled “j48” in table 4.10). Second, we consider the j48 algorithm, but we multiply the number of instances by the cost of each instance, to create a cost sensitive decision tree (labelled “cost sensitive”). Third, we use the cost sensitive approach, but using only the subset of attributes selected by the CFS algorithm (labelled “cost sensitive, CFS”). Fourth, we use the cost sensitive decision tree built using only the attributes in the Expert subset (labelled “cost sensitive, Expert”). Finally, we consider the cost sensitive decision tree, built using the Expert subset and with the following modification to the j48 algorithm (labelled “cost sensitive Expert, pruned”). The j48 algorithm is parametrized to allow manipulation of the number of data instances represented by a leaf node in the tree. This value is set at 2 by default, indicating that there must be at least 2 data points matching the criteria for the leaf, or that the leaf node is pruned and the parent node becomes a leaf. This is done to reduce over fitting the decision tree to the data. However, there are single instances that when we multiply the number of instances by their importance, produce enough instances to form a leaf in the tree. To compensate, we increase the number of data points per leaf from 2 to 2 times the mean cost for the dataset. Thus, for the All Problems Dataset, 4114 instances were required per leaf, 22112 for the Random Problems Dataset, 2244 for the Small World Problems Dataset, and 3528 for the QWH Problems Dataset. The table shows that the cost sensitive versions tend to significantly improve the total misclassification cost over the non-cost sensitive algorithm. Using the attribute subset chosen by CFS is quite good for the Random Problems Dataset, but not very good for the Small World Problems Dataset. The increased number of data points per leaf is very effective for the Small World Problems Dataset and Random Problems Dataset, but not very effective for the QWH Problems Dataset. We suspect that the poor performance on the QWH Problems Dataset is due to the distribution of AC and FC instances, with respect to cost. Referring to Figure 3.12, we see that there are only AC instances with high cost, so the algorithm will be biased to ignore FC instances.

To summarize, we can reduce the total misclassification costs of the decision trees by weighting each instance according to its relative importance. Furthermore, inducing greater pruning of leaf nodes in the decision tree greatly improves the performance in all but one of the cost sensitive datasets.

4.4 Classifying Previously Unseen Problem Types

In this section, we investigate the usefulness of a classifier trained on one class of problems and then applied to a different class of problems. Whereas we usually assume our training and test sets are distinct, but drawn from the same class of problem, here we do something slightly different. We have trained a decision tree using all attributes and a non-cost sensitive approach. We trained the

Table 4.10: Total misclassification cost (milliseconds) for each algorithm. j48 is the standard j48 algorithm using all 24 attributes, cost sensitive uses all attributes with cost sensitive classification, cost sensitive CFS uses the CFS variable selection. The last two columns are cost sensitive and use the Expert attribute subset, and the pruned column includes greater pruning of the j48 tree as described in the text.

	j48	cost sensitive	cost sensitive, CFS	cost sensitive, Expert	cost sensitive, Expert,pruned
Random	1 302 942	467 023	244 138	334 085	272 713
Small World	12 462 649	6 100 256	9 407 976	5 437 703	2 133 446
QWH	3 278	1 588	1 524	1 520	3 833
All	14 328 691	9 254 939	10 631 129	6 683 594	2 151 305

decision tree using each of the three distinct data sets (Random Problems, Small World Problems, QWH Problems) and in each case tested on the other two datasets. The results are presented in Table 4.11. For comparison, we have included the results of training and testing on like problems, repeated from Tables 4.5-4.7. As in the previous tables, we present the accuracy percentage with κ values in parentheses.

We see the best results when training on the Random Problems Dataset and testing on the Small World Problems Dataset, with a κ value of 0.2519. The worst results occur when training on the QWH Problems Dataset and testing on the Small World Problems Dataset. This suggests that the information gained by studying random problems provides an insight into small world problems, with a 25% improvement in prediction accuracy over chance agreement. However, training on QWH problems provides incorrect insights into small worlds, indicating that the advice that is correct for QWH problems is counter to the advice for small world prbolems. It would be interesting in future work to examine how many examples from the test class are needed to improve accuracy. For example, how well could a large set of random problems and a small sample of small worlds be used to accurately make predictions about small worlds? Additionally, how well can information from small problems be used to make predictions about large problems?

In summary, as we expected, performance was generally poor when the training and testing sets are known to be dissimilar.

4.5 Usability of Metareasoning for CSPs

After assessing the quality of the metareasoning CSP solver, we now examine the practicality of using such a solver. The costs involved in using a metareasoning CSP solver are: the time required to collect the problem attributes used for metareasoning (MR), the time to reason about which

Table 4.11: Prediction accuracy (%) using separate testing and training classes, κ values presented in parentheses.

		training set					
		Random		Small World		QWH	
test set	Random	88.18	(0.5336)	64.87	(0.123)	68.58	(-0.074)
	Small World	68.16	(0.2519)	78.85	(0.5218)	43.61	(-0.2636)
	QWH	74.83	(-0.066)	79.02	(0)	91.45	(0.7245)

solver to apply (t_r), and the time to solve using the predicted solver (t_p). The decision trees we have created are generally quite small, up to 10 levels deep, and could be compiled to native code, resulting in a short series of conditional checks to arrive at a prediction. Therefore, we assume that the time to consult the decision tree is negligible ($t_r = 0$).

In order to determine if metareasoning is practical, we must calculate MR and t_p. For some problems, we expect that the time to compute the attributes used for metareasoning may be longer than the time to solve the problem. These cases may occur when the problems are trivial. Conversely, for very hard problems, the time to compute the attributes used for metareasoning may be insignificant relative to the time to solve the problem.

First, we consider how long will it take to collect the problem attributes needed to metareason about a problem. We have recorded the time required to collect the attributes for each of the problems under consideration and averaged over 10 runs. All of the methods used for attribute collection are deterministic, so all the variance in time is due to noise caused by the operating system or other processes running simultaneously. We have chosen to measure how long it takes to collect all the attributes we have considered, as well as the Expert subset. Future work may weigh the cost and benefit of collecting each individual attribute.

In Figure 4.1, we can see the relationship between the size of a problem instance, measured by the number of variables and the amount of time to collect the attributes for metareasoning. It is clear that the length of time required to collect just a subset of the attributes is significantly smaller as the size of the problem instance increases. However, for smaller problems, the advantage is much smaller. Figure 4.1 contains plots for collecting all attributes and the Expert attribute subset. It is clear that the time to collect the attribute subset increases much more slowly with respect to n, than collecting all attributes. Given that we have seen in Tables 4.4-4.7 that the prediction accuracy is roughly the same using all attributes or the attribute subset, we can conclude that using a small subset of attributes will be much more effective for large problems due to the time saved on collecting the attributes.

In order to determine t_p, we can calculate the expected cost of a metareasoning CSP solver by

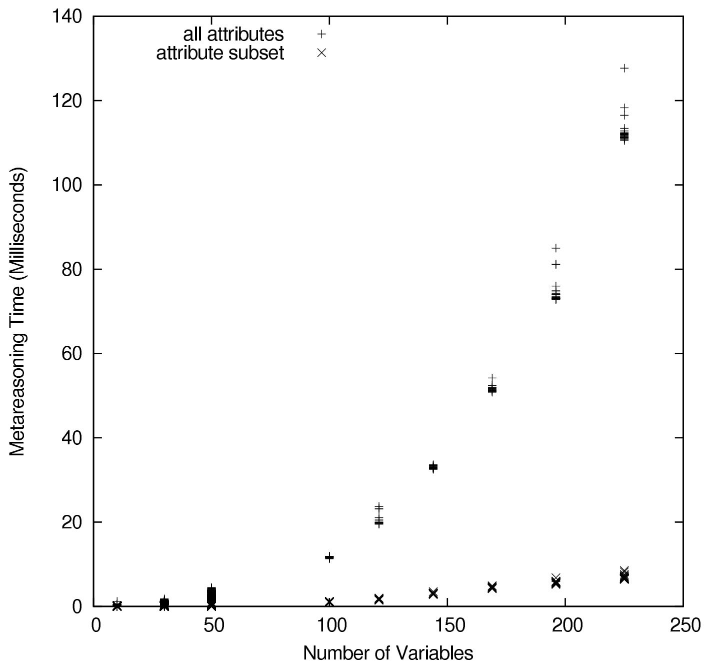


Figure 4.1: Time to collect attributes for metareasoning (MR time) vs number of variables

summing the expected time to solve a CSP using the predicted method. The expected time to solve a problem with the method predicted by the metareasoner is

$$t_p = p(right) * t_{right} + p(wrong) * t_{wrong}, \tag{4.1}$$

where $p(right)$ is the probability of a correct classification, t_{right} is the time to solve a problem with the correct solver, $p(wrong)$ is the probability of making a wrong classification, and t_{wrong} is the time to solve a problem with the wrong solver. The expected cost of the metareasoning solver is $MR + t_p$. An oracle solves problems in time t_* on average, Arc Consistency solves problems in time t_{ac} on average, and Forward Checking solves problems in time t_{fc} on average. If $(t_p > t_{ac}) \vee (t_p > t_{fc})$, then the metareasoning solver should not be used as one of the two basic solvers has better average performance than the metareasoning solver. For each problem instance, we determine the total solving time by summing the metareasoning cost and the time to solve using the predicted solver.

In Table 4.12 we can see the average time required to solve problems in the all problems in each of the 4 datasets using FC, AC, an oracle, an anti-oracle (a solver that always makes the wrong choice), and two variants of the metareasoning CSP solver. Each entry in the table is the average value over the problem set. The first metareasoning solver, $MR + t_p$, uses all attributes and is not cost sensitive. The second, $MR_s + t_{ps}$, uses only the attributes in the Expert subset, is cost sensitive, and uses increased pruning of the decision tree as described in section 4.3. $MR + t_p$ represents the combined cost of collecting all attributes, applying the non-cost sensitive decision tree to select an algorithm, and running either AC or FC, whichever is selected. $MR_s + t_{ps}$ is the

Table 4.12: Average time to solve all problems from 4 datasets. t_{fc} and t_{ac} use Forward Checking and Arc Consistency, respectively. t_* and t_{worst} are an oracle that always chooses correctly between AC and FC and an anti-oracle that always chooses the slower of AC and FC. $MR + t_p$ corresponds to first metareasoner presented (basic j48 algorithm, all attributes, non-cost sensitive. $MR_s + t_{ps}$ corresponds to the improved metareasoner (j48 algorithm, cost sensitive, increased pruning, Expert attribute subset).

	t_{fc}	t_{ac}	t_*	t_{worst}	$MR + t_p$	$MR_s + t_{ps}$
All	2903	3363	2104	4161	2375	2145
Random	11709	21651	11152	22209	11420	11208
Small World	2015	1586	1240	2361	1509	1286
QWH	2072	317	313	2076	376	319

Table 4.13: Average time to solve all problems from 4 datasets by metareasoning, decomposed into solving time and metareasoning time. $MR + t_p$, t_p, and MR are the combined, and split, times to solve using the first metareasoner presented (basic j48 algorithm, all attributes, non-cost sensitive. $MR_s + t_{ps}$, t_{ps}, and MR_s are the combined and split times to solve with the improved metareasoner (j48 algorithm, cost sensitive, increased pruning, Expert attribute subset).

	t_p	MR	$MR + t_p$	t_{ps}	MR_s	$MR_s + t_{ps}$
All	2371	3.79	2375	2145	0.29	2145
Random	11418	1.89	11420	11208	0.24	11208
Small World	1508	1.10	1509	1286	0.09	1286
QWH	315	61.13	376	315	4.46	319

amount of time required to collect the Expert attribute subset, applying the cost sensitive decision tree, and running either AC or FC, whichever is selected. Table 4.13 presents the amount of time spent solving and time spent metareasoning for the two metareasoning variants.

We can see that, generally, FC is preferred on the Random Problems Dataset and AC is preferred on the Small World Problems Dataset. AC is preferred on the QWH Problems Dataset instances with significant runtime. The optimized metareasoner has much lower expected cost than the non-optimized version, with the exception of the QWH Problems Dataset. As we mentioned in the discussion of cost sensitive classification, raising the number of training instances per leaf had a negative impact on the QWH Problems Dataset because nearly all the problems with large cost are AC problem instances.

In Tables 4.14-4.17 we present the 95% confidence intervals for the difference in the mean time to solve all problems in our samples. From these tables, we can see that the difference in mean solving time is significant in all but three cases, which are highlighted in bold. Negative values

Table 4.14: 95% confidence intervals for difference in mean time in milliseconds (All Problems Dataset)

	t_{ac}	t_*	t_{worst}	$MR + t_p$	$MR_s + t_{ps}$
t_{fc}	-459 ± 264	799 ± 177	-1258 ± 196	528 ± 138	758 ± 176
t_{ac}		1258 ± 196	-799 ± 177	988 ± 225	1218 ± 196
t_*			-2057 ± 264	-270 ± 112	-40 ± 18
t_{worst}				1786 ± 239	2016 ± 263
$MR + t_p$					230 ± 111

Table 4.15: 95% confidence intervals for difference in mean time in milliseconds (QWH Problems Dataset)

	t_{ac}	t_*	t_{worst}	$MR + t_p$	$MR_s + t_{ps}$
t_{fc}	1755 ± 1037	1759 ± 1037	-4 ± 0	1697 ± 1037	1753 ± 1037
t_{ac}		4 ± 0	-1759 ± 1037	-58 ± 1	-2 ± 1
t_*			-1764 ± 1037	-63 ± 2	-6 ± 1
t_{worst}				1701 ± 1037	1757 ± 1037
$MR + t_p$					56 ± 1

indicate that the solver on the left hand side is faster than the solver on the top. Conversely, positive values indicate the solver on the top of the table is faster than the one on the side.

In Figures 4.2-4.5 we plot the fraction of problems solved and a cost, relative to solving the problems with an oracle. A data point on the figure indicates the fraction of problems that each particular solver can solve within a specified margin of the oracle solver. Thus, from Figure 4.2, we can see that the metareasoning solver can solve approximately 85% of the All Problems Dataset within 1 second of the time taken by an oracle solver. For the All Problems Dataset, the metareasoning solver outperforms AC and FC for margins greater than 1ms. In the Random Problems Dataset (Figure 4.3), FC is shown to solve over 90% of the problems within 1ms of the time taken by an oracle. However, the metareasoning solver performs slightly better than FC for margins greater than 2ms. In the Small World Problems Dataset (Figure 4.4), the metareasoning solver is superior to AC and FC for margins greater than 1ms. Finally, in the QWH Problems Dataset (Figure 4.5) the metareasoning solver is the preferred solver for margins between 6 and 10ms. Before which, FC is preferred and after which AC is preferred. We would expect that the metareasoning solver should outperform both AC and FC, but the sharp transition from problems that should be solved with AC to problems that should be solved with FC, as seen in Figure 3.12, is not detected by the

Table 4.16: 95% confidence intervals for difference in mean time in milliseconds (Random Problems Dataset)

	t_{ac}	t_*	t_{worst}	$MR + t_p$	$MR_s + t_{ps}$
t_{fc}	−9942 ± 2089	557 ± 416	−10499 ± 2044	290 ± 201	501 ± 415
t_{ac}		10499 ± 2044	−557 ± 416	10231 ± 2078	10443 ± 2045
t_*			−11056 ± 2084	**−267 ± 376**	−55 ± 40
t_{worst}				10789 ± 2051	11001 ± 2084
$MR + t_p$					**212 ± 376**

Table 4.17: 95% confidence intervals for difference in mean time in milliseconds (Small World Problems Dataset)

	t_{ac}	t_*	t_{worst}	$MR + t_p$	$MR_s + t_{ps}$
t_{fc}	430 ± 203	776 ± 193	−346 ± 61	507 ± 149	730 ± 192
t_{ac}		346 ± 61	−776 ± 193	**77 ± 137**	300 ± 64
t_*			−1122 ± 202	−269 ± 122	−46 ± 24
t_{worst}				852 ± 161	1076 ± 201
$MR + t_p$					223 ± 122

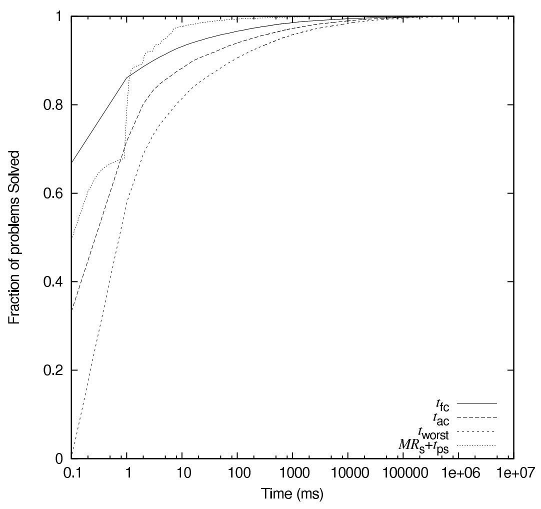


Figure 4.2: Runtime distribution: All Problems Dataset

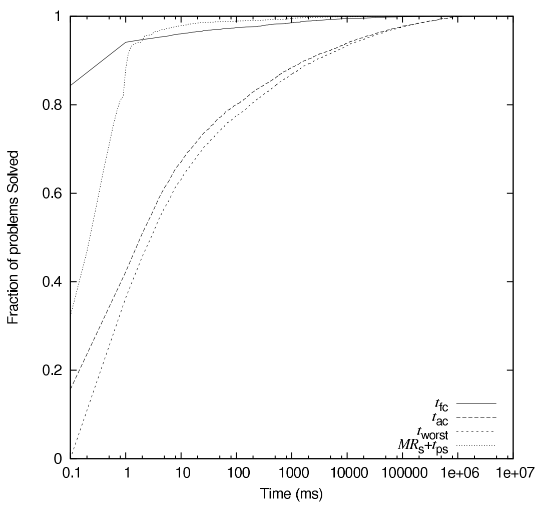


Figure 4.3: Runtime distribution: Random Problems Dataset

decision tree.

In summary, the Metareasoning CSP solver is significantly faster than either AC or FC on the All Problems Dataset, Random Problems Dataset, and Small World Problems Dataset. However, Arc Consistency is the preferred algorithm on the QWH Problems Dataset.

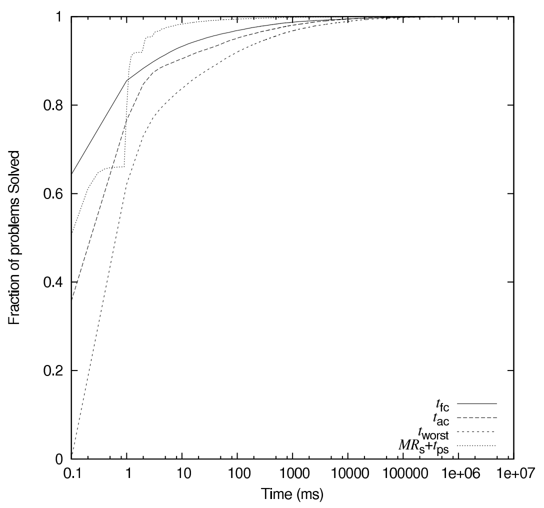


Figure 4.4: Runtime distribution: Small World Problems Dataset

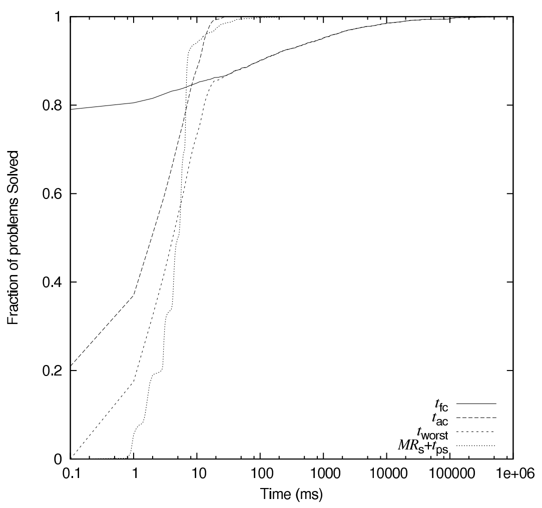


Figure 4.5: Runtime distribution: QWH Problems Dataset

Chapter 5
Conclusions

Given that constraint satisfaction is a general problem solving framework, performance improvements in CSP solving can benefit a large number of application domains. In this work we explored the use of metareasoning to select a problem solving method believed to be appropriate to the current problem, based on past experience.

5.1 Contributions

We selected a total of 24 attributes to record for each CSP instance. We solved each CSP instance using both Forward Checking and Arc Consistency and recorded how long each method took to solve the problem. We then trained decision trees to classify problem instances as either being solved faster with FC or AC. We were able to distinguish when to apply AC or FC with accuracy between 79% and 98%, depending on the problem set.

We were able to show that a small subset of attributes can be used to classify the instances with high accuracy, often as well or better than using the whole attribute set. Furthermore, we found that some attributes are useful for one type of problem, but not for another, and some attributes are useful for distinguishing between the problem classes.

As we would intuitively expect, the prediction accuracy rates tend to rise as we consider subsets of the problems where there is a larger difference in the runtime of AC and FC.

Some problem instances have much higher differences in AC and FC solving time than others. As such, we are interested in minimizing the overall solving time, rather than maximizing the number of instances correctly classified. We increased the relative importance of classification on instances with large differences in the runtime of AC and FC and found we could significantly reduce the average time to solve problems over the dataset.

We examined the performance of the decision trees when training and testing on datasets known to be drawn from different distributions. For some combinations of training and testing sets, we observed moderate success in the classification task, while on others we observed fewer correct classifications than expected by chance. In some cases, knowledge gained on one class of problem can provide some information about a new class of problem. However, because the decision trees are trained to classify one particular type of problem, they may be ignoring the attributes that are useful in classifying another group of problems. An alternative learning scheme, such as KNN may have been more successful for this task.

Finally, we assessed the overhead of metareasoning by measuring the amount of time required to collect the problem attributes needed. For the Random Problems Dataset and Small World Problems Dataset, we observed that the time to collect the problem attributes was minimal compared to the time to solve the problem. For the QWH Problems Dataset, the attribute collection was found to be non-negligible. When using only a subset of the attributes for metareasoning, we were able to significantly reduce the amount of time required to collect the attributes. When we summed together the time to collect the problem attributes, and the expected time to solve the CSPs, given the error rates of the classifier, we observed a significant improvement over using either AC of FC in most cases. Specifically, we saw significant improvements in the All, Random, and Small Worlds datasets and performance was slightly worse than AC alone on the QWH dataset. Thus, we can conclude that, for some datasets, choosing a solving algorithm to suit each specific problem can be significantly faster than choosing a single algorithm for all problems.

5.2 Directions for Future Work

In this work we have presented a classifier that chooses between two classes; however, in practice, we may have hundreds, or thousands, of parametrizations. Thus, future work should find more effective learning methods for addressing this general classification problem. One problem with extending the current method is that there is no categorization or ranking of solvers. One solution may be to find methods of ranking solvers, perhaps by how much propagation they perform. Thus, intermediate propagators between AC and FC can be treated by learning algorithms as intermediate methods, rather than treating all propagators as different but equal. The advantage to this technique is that if AC is the correct classification, then an intermediate propagator can be recognized as a lesser mistake than choosing FC. However, the ordering of the propagators may be dependent on the class of problem being solved, which would mean a new ordering would be needed for each class of problems. Alternatively, rather than classifying instances, we could use regression to predict the time to solve an instance with each method and then select the method with the smallest expected time to solve, as was done in L. Xu et al. (2008). Regression has a few benefits over classification. For example, as the output of the learning algorithm is a model for the runtime of a particular solver, we do not need to use exactly the same training set for each solver. This would allow the training data to be generated by solving real problems, without the need to resolve each problem with alternate solvers as is needed for classification. However, by not using a fixed training set for modelling each solver, we may produce a very good model of one solver while producing a very poor model of another. Furthermore, the costs of metareasoning may be higher for regression than classification, as we would need to consult multiple models to determine the expected runtime of all candidate solvers rather than consulting a single classifier.

Finally, looking beyond algorithm selection, we expect that there may be sub-problems within a CSP that are better suited to one solving method over another. As such, we are interested in changing solvers mid-problem, which is an on demand algorithm construction problem.

We will continue to expand the set of candidate solvers to be considered, by implementing a variety of backtracking techniques, adding additional variable and value heuristics, and additional levels of consistency. In addition, we will extend our approach to other parametrized CSPs with non-random constraints and non-random constraint graphs, as well as CSPs drawn from the real world. Lastly, we will record more attributes of each CSP, in order to improve the prediction accuracy of the learned model, which will provide a richer set of data to analyze for patterns.

References

Achlioptas, D., Gomes, C., Kautz, H., & Selman, B. (2000). Generating satisfiable problem instances. In *Proceedings of the 17th National Conference on Artificial Intelligence* (pp. 256–261).

Aha, D., Kibler, D., & Albert, M. (1991). Instance-based learning algorithms. *Machine Learning*, *6*(1), 37–66.

Balafoutis, T., & Stergiou, K. (2008a). Experimental evaluation of modern variable selection strategies in constraint satisfaction problems. In *CEUR Workshop Proceedings: Proceedings of the 15th RCRA Workshop on Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion* (Vol. 451). Available from `http://tmancini.di.uniroma1.it/rcra/workshops/RCRA-2008/Accepted_papers_files/Balafoutis-Stergiou-RCRA2008-P20.pdf`

Balafoutis, T., & Stergiou, K. (2008b). Exploiting constraint weights for revision ordering in Arc Consistency algorithms. In *Proceedings of the ECAI 2008 Workshop on Modeling and Solving Problems with Constraints* (pp. 1–7).

Bartak, R. (2001). Theory and practice of constraint propagation. In *Proceedings of the Third Workshop on Constraint Programming in Decision and Control* (pp. 7–14).

Beck, J., Prosser, P., & Wallace, R. (2004). Variable ordering heuristics show promise. In *LNCS 3258: Proceedings of the 10th International Conference on Principles and Practice of Constraint Programming* (pp. 711–715).

Beck, J., Prosser, P., & Wallace, R. (2005). Trying again to fail-first. *LNAI 3419: Recent Advances in Constraints*, 41–55.

Bessière, C., Chmeiss, A., & Sais, L. (2001). Neighborhood-based variable ordering heuristics for the constraint satisfaction problem. In *LNCS 2239: Proceedings of the Seventh International Conference on Principles and Practice of Constraint Programming* (pp. 565–569).

Bessière, C., & Régin, J. (1996). MAC and combined heuristics: Two reasons to forsake FC (and CBJ?) on hard problems. In *LNCS 1118: Proceedings of the Second International Conference on Principles and Practice of Constraint Programming* (p. 61-75).

Bessière, C., Régin, J., Yap, R., & Zhang, Y. (2005). An optimal coarse-grained arc consistency algorithm. *Artificial Intelligence*, *165*(2), 165–185.

Bittle, S., & Fox, M. (2009). Learning and using hyper-heuristics for variable and value ordering in constraint satisfaction problems. In *Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference* (pp. 2209–2212).

Boussemart, F., Hemery, F., Lecoutre, C., & Sais, L. (2004). Boosting systematic search by weighting constraints. In *Proceedings of the 16th European Conference on Artificial Intelligence.* Available from `http://www.frontiersinai.com/ecai/ecai2004/ecai04/pdf/p0146.pdf`

Cabon, B., De Givry, S., Lobjois, L., Schiex, T., & Warners, J. (1999). Radio link frequency assignment. *Constraints*, *4*(1), 79–89.

Cambazard, H., & Jussien, N. (2006). Identifying and exploiting problem structures using explanation-based constraint programming. *Constraints*, *11*(4), 295–313.

Cheeseman, P., Kanefsky, B., & Taylor, W. (1991). Where the really hard problems are. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence* (pp. 331–337).

Cohen, J. (1960). A coefficient of agreement for nominal scales. *Educational and Psychological Measurement*, *20*(1), 37–46.

Cox, M., & Raja, A. (2008). Metareasoning: A manifesto. In *Proceedings of AAAI 2008 Workshop on Metareasoning: Thinking about Thinking* (pp. 1–4).

Day, D. (1992). Acquiring search heuristics automatically for constraint-based planning and scheduling. In *Proceedings of the First International Conference on Artificial Intelligence Planning Systems* (pp. 45–51).

Dechter, R. (1986). Learning while searching in constraint-satisfaction problems. In *Proceedings of the Fifth National Conference on Artificial Intelligence.* Available from `www.aaai.org/Papers/AAAI/1986/AAAI86-029.pdf`

Dechter, R. (2003). *Constraint Processing.* San Francisco: Morgan Kaufmann.

El Sakkout, H., Wallace, M., & Richards, E. (1996). An instance of adaptive constraint propagation. In *LNCS 1118: Proceedings of the Second International Conference on Principles and Practice of Constraint Programming* (pp. 164–178).

Epstein, S. (1992). Prior knowledge strengthens learning to control search in weak theory domains. *International Journal of Intelligent Systems*, *7*(6), 547–586.

Epstein, S. (2009). Integrating a portfolio of representations to solve hard problems. In *In Proceedings of the AAAI Fall Symposium on Multi-Representational Architectures for Human-Level Intelligence* (pp. 8–13).

Epstein, S., & Petrovic, S. (2007). Learning to solve constraint problems. In *Proceedings of the 17th International Conference on Automated Planning and Scheduling: Workshop on Planning and Learning.* Available from `http://www.cs.umd.edu/~ukuter/icaps07aipl/proceedings/paper11.pdf`

Epstein, S., Wallace, R., Freuder, E., & Li, X. (2005). Learning propagation policies. *Proceedings of the Second International Workshop on Constraint Propagation And Implementation*, 1–15.

Estlin, T., & Mooney, R. (1997). Learning to improve both efficiency and quality of planning. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence* (pp. 1227–1233).

Fayyad, U., & Irani, K. (1993). Multi-interval discretization of continuous attributes as preprocessing for classification learning. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence* (pp. 1022–1027).

Fern, A., Yoon, S., & Givan, R. (2004). Learning domain-specific control knowledge from random walks. In *Proceedings of the 14th International Conference on Automated Planning and Scheduling* (pp. 191–199).

Freuder, E. (1982). A sufficient condition for backtrack-free search. *Journal of the ACM*, *29*(1), 24–32.

Frost, D., & Dechter, R. (1995). Look-ahead value ordering for constraint satisfaction problems. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence* (pp. 572–578).

Gebruers, C., Hnich, B., Bridge, D., & Freuder, E. (2005). Using CBR to select solution strategies in constraint programming. In *LNCS 3620: Proceedings of the Sixth International Conference on Case-Based Reasoning* (pp. 222–236).

Gent, I., MacIntyre, E., Prosser, P., Smith, B., & Walsh, T. (2001). Random constraint satisfaction: Flaws and structure. *Constraints*, *6*(4), 345–372.

Gent, I., MacIntyre, E., Prosser, P., & Walsh, T. (1995). Scaling effects in the csp phase transition. In *LNCS 976: Proceedings of the First International Conference on Principles and Practice of Constraint Programming* (pp. 70–87).

Gent, I., MacIntyre, E., Prosser, P., & Walsh, T. (1996). The constrainedness of search. In *Proceedings of the 13th National Conference on Artificial Intelligence* (pp. 246–252).

Gomes, C., & Shmoys, D. (2002). Completing quasigroups or latin squares: A structured graph coloring problem. In *Proceedings of the Computational Symposium on Graph Coloring and Generalizations* (pp. 22–39).

Good, I. (1971). Twenty-seven principles of rationality. In V. Godambe & D. Sprott (Eds.), *Foundations of statistical inference* (pp. 108–141). Holt, Rinehart and Winston, Toronto.

Gordon, A., Hobbs, J., & Cox, M. (2008). Anthropomorphic self-models for metareasoning agents.

In *Proceedings of AAAI 2008 Workshop on Metareasoning: Thinking about Thinking* (pp. 129–135).

Haim, S., & Walsh, T. (2008). Online estimation of SAT solving runtime. In *Proceedings of the 11th International Conference on Theory and Applications of Satisfiability Testing* (pp. 133–138).

Hall, M. (2000). Correlation-based feature selection for discrete and numeric class machine learning. In *Proceedings of the 17th International Conference on Machine Learning* (pp. 359–366).

Haralick, R., & Elliott, G. (1980). Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, *14*(3), 263–313.

Holte, R. (1993). Very simple classification rules perform well on most commonly used datasets. *Machine Learning*, *11*(1), 63–90.

Horvitz, E., Ruan, Y., Gomes, C., Kautz, H., Selman, B., & Chickering, M. (2001). A Bayesian approach to tackling hard computational problems. In *Proceedings the 17th Conference on Uncertainty in Artificial Intelligence* (pp. 235–244).

Huang, Y., Selman, B., & Kautz, H. (2000). Learning declarative control rules for constraint-based planning. In *Proceedings of 17th International Conference on Machine Learning* (pp. 415–422).

Hutter, F., Hamadi, Y., Hoos, H., & Leyton-Brown, K. (2006). Performance prediction and automated tuning of randomized and parametric algorithms. *LNCS 4204: Proceedings of the 12th International Conference on Principles and Practice of Constraint Programming*, 213–228.

Hutter, F., Hoos, H., Leyton-Brown, K., & Stützle, T. (2009). ParamILS: An automatic algorithm configuration framework. *Journal of Artificial Intelligence Research*, *36*(1), 267–306.

Hutter, F., Hoos, H., & Stützle, T. (2007). Automatic algorithm configuration based on local search. In *Proceedings of the 22nd conference on artificial intelligence* (pp. 1152–1157).

Jeroslow, R., & Wang, J. (1990). Solving propositional satisfiability problems. *Annals of Mathematics and Artificial Intelligence*, *1*(1), 167–187.

John, G., & Langley, P. (1995). Estimating continuous distributions in Bayesian classifiers. In *Proceedings of the 11th Conference on Uncertainty in Artificial Intelligence* (pp. 338–345).

Jones, J., & Goel, A. (2009). Metareasoning for adaptation of classification knowledge. In *Proceedings of The Eighth International Conference on Autonomous Agents and Multiagent Systems* (pp. 1145–1146).

Laird, J., Newell, A., & Rosenbloom, P. (1987). Soar: An architecture for general intelligence. *Artificial Intelligence*, *33*(1), 1–64.

Lecoutre, C., Boussemart, F., & Hemery, F. (2003). Exploiting multidirectionality in coarse-grained arc consistency algorithms. *LNCS 2833: Proceedings of the 9th International Conference on Principles and Practice of Constraint Programming*, 480–494.

Lecoutre, C., & Hemery, F. (2007). A study of residual supports in arc consistency. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence* (pp. 125–130).

Lecoutre, C., Sais, L., & Vion, J. (2007). Using SAT Encodings to Derive CSP Value Ordering Heuristics. *Journal on Satisfiability, Boolean Modeling and Computation*, *1*, 169–186.

Likitvivatanavong, C., Zhang, Y., Shannon, S., Bowen, J., & Freuder, E. (2007). Arc consistency during search. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence* (pp. 137–142).

Long, D., & Fox, M. (2003). The 3rd international planning competition: Results and analysis. *Journal of Artificial Intelligence Research*, *20*(1), 1–59.

Mackworth, A. (1977a). Consistency in networks of relations. *Artificial Intelligence*, *8*(1), 99–118.

Mackworth, A. (1977b). On reading sketch maps. In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence* (p. 598-606).

Minton, S. (1996). Automatically configuring constraint satisfaction programs: A case study. *Constraints*, *1*(1), 7–43.

Morbini, F., & Schubert, L. (2008). Metareasoning as an integral part of commonsense and autocognitive reasoning. In *Proceedings of AAAI 2008 Workshop on Metareasoning: Thinking about Thinking* (pp. 155–162).

Nadel, B. (1990). Constraint satisfaction algorithms. *Computational Intelligence*, *5*(4), 188–224.

Petrovic, S., & Epstein, S. (2008). Random Subsets Support Learning a Mixture of Heuristics. *International Journal on Artificial Intelligence Tools*, *17*(3), 501–520.

Pulina, L., & Tacchella, A. (2009). A self-adaptive multi-engine solver for quantified Boolean formulas. *Constraints*, *14*(1), 80–116.

Quinlan, J. (1986). Induction of decision trees. *Machine Learning*, *1*(1), 81–106.

Quinlan, J. (1990). Learning logical definitions from relations. *Machine learning*, *5*(3), 239–266.

Quinlan, J. (1993). *C4. 5: programs for machine learning.* San Mateo, CA: Morgan Kaufmann.

Refalo, P. (2004). Impact-based search strategies for constraint programming. *LNCS 3258: Proceedings of the 10th International Conference on Principles and Practice of Constraint Programming*, 557–571.

Russell, S. (1991). An architecture for bounded rationality. *ACM SIGART Bulletin*, *2*(4), 146–150.

Russell, S., & Norvig, P. (2003). *Artificial Intelligence: A Modern Approach.* Upper Saddle River, NJ: Prentice-Hall.

Sabin, D., & Freuder, E. (1994). Contradicting conventional wisdom in constraint satisfaction. In *Proceedings of the 11th European Conference on Artificial Intelligence* (pp. 125–129).

Shannon, C., & Weaver, W. (1948). The mathematical theory of communication. *Bell System Technical Journal*, *27*, 379–423.

Smith, B. (1994). The phase transition in constraint satisfaction problems: A closer look at the mushy region. In *Proceedings of the 11th European Conference on Artificial Intelligence* (pp. 100–104).

Stamatatos, E., & Stergiou, K. (2009). Learning how to propagate using random probing. In *Proceedings of the 6th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems* (pp. 263–278).

Stergiou, K. (2008). Heuristics for dynamically adapting propagation. In *Proceedings of the 18th European Conference on Artificial Intelligence* (pp. 485–489).

Tsang, E. (1993). *Foundations of Constraint Satisfaction.* San Diego, CA: Academic Press.

Tsoumakas, G., Vrakas, D., Bassiliades, N., & Vlahavas, I. (2004). Using the k-nearest problems for adaptive multicriteria planning. In *LNAI 3025: Methods and Applications of Artificial Intelligence: Proceedings of the Third Hellenic Conference on AI* (pp. 132–141).

Ulam, P., Jones, J., & Goel, A. (2008). Combining model-based meta-reasoning and reinforcement learning for adapting game-playing agents. In *Proceedings of the Fourth Artificial Intelligence and Interactive Digital Entertainment Conference.* Available from `www.aaai.org/Papers/AIIDE/2008/AIIDE08-022.pdf`

Wallace, M., & Schimpf, J. (2002). Finding the right hybrid algorithm–A combinatorial meta-problem. *Annals of Mathematics and Artificial Intelligence*, *34*(4), 259–269.

Wallace, R., & Grimes, D. (2008). Experimental studies of variable selection strategies based on constraint weights. *Journal of Algorithms*, *63*(1-3), 114–129.

Watts, D., & Strogatz, S. (1998). Collective dynamics of 'small-world' networks. *Nature*, *393*, 440–442.

Witten, I. H., & Frank, E. (2005). *Data Mining: Practical Machine Learning Tools and Techniques* (2nd ed.). San Francisco, CA: Morgan Kaufmann.

Xu, K., & Li, W. (2000). Exact phase transitions in random constraint satisfaction problems. *Journal of Artificial Intelligence Research*, *12*, 93–103.

Xu, L., Hutter, F., Hoos, H., & Leyton-Brown, K. (2008). SATzilla: portfolio-based algorithm selection for SAT. *Journal of Artificial Intelligence Research*, *32*(1), 565–606.

Xu, L., Hutter, F., Hoos, H., & Leyton-Brown, K. (2009). *SATzilla2009: an automatic algorithm portfolio for SAT.* Available from `http://www.cs.ubc.ca/labs/beta/Projects/SATzilla/SATzilla2009.pdf`

Xu, Y., Stern, D., & Samulowitz, H. (2009). Learning adaptation to solve constraint satisfaction problems. *Learning and Intelligent OptimizatioN (LION)*. Available from `http://lion.disi.unitn.it/intelligent-optimization//LION3/online_proceedings/56.pdf`

Zanarini, A., & Pesant, G. (2009). Solution counting algorithms for constraint-centered search heuristics. *Constraints*, *14*(3), 392–413.

Zhang, W. (2001). Phase transitions and backbones of 3-SAT and maximum 3-SAT. In *LNCS 2239:*

Proceedings of the Seventh International Conference on Principles and Practice of Constraint Programming (pp. 153–167).

Zheng, J., & Horsch, M. (2005). A Decision Theoretic Meta-reasoner for Constraint Optimization. In *LNCS 3501: Advances in Artificial Intelligence: Proceedings of the 18th Conference of the Canadian Society for Computational Studies of Intelligence* (pp. 53–65).