# Evolving Algorithms for Over-Constrained and Satisfaction Problems

by

## Stuart Bain

**BInfTech BEng(Hons)**

School of Information and Communication Technology

Faculty of Engineering and Information Technology

Griffith University, Queensland

Australia

A thesis submitted in fulfillment

of the requirements of the degree of

Doctor of Philosophy

November, 2006

# Statement of Originality

This work has not been previously submitted for a degree or diploma in any university. To the best of my knowledge and belief, the thesis contains no material previously published or written by another person except where due reference is made in the thesis itself.

<div align="right">

_____

*Stuart Iain Bain*

</div>

"In my experience, however, the way to have the best chance of discovering new phenomena in a computer experiment is to make the design of the experiment as simple and direct as possible. It is usually much better, for example, to do a mindless search of a large number of possible cases than to do a carefully crafted search of a smaller number. For in narrowing the search one inevitably makes assumptions, and these assumptions may end up missing the cases of greatest interest."

Stephen Wolfram (2002) *"A New Kind of Science"*, page 111.

# Abstract

The notion that a universally effective problem solver may still exist, and is simply waiting to be found, is slowly being abandoned in the light of a growing body of work reporting on the narrow applicability of individual heuristics. As the formalism of the *constraint satisfaction problem* remains a popular choice for the representation of problems to be solved algorithmically, there exists an ongoing need for new algorithms to efficiently handle the disparate range of problems that have been posed in this representation. Given the costs associated with manually applying human algorithm development and problem solving expertise, methods that can automatically adapt to the particular features of a specific class of problem have begun to attract more attention.

Whilst a number of authors have developed adaptive systems, the field, and particularly with respect to their application to constraint satisfaction problems, has seen only limited discussion as to what features are desirable for an adaptive constraint system. This may well have been a limiting factor with previous implementations, which have exhibited only subsets of the *five features identified in this work as important to the utility of an adaptive constraint satisfaction system.*

Whether an adaptive system exhibits these features depends on both the chosen representation and the method of adaptation. In this thesis, *a three-part representation for constraint algorithms* is introduced, which defines an algorithm in terms of contention, preference and selection functions. An *adaptive system based on genetic programming* is presented that adapts constraint algorithms described using the mentioned three-part representation. This is believed to be the first use of standard genetic programming for learning constraint algorithms.

Finally, to further demonstrate the efficacy of this adaptive system, its performance in *learning specialised algorithms for hard, real-world problem instances* is thoroughly evaluated. These instances include random as well as structured instances from known-hard benchmark distributions, industrial problems (specifically, SAT-translated planning and

cryptographic problems) as well as over-constrained problem instances. The outcome of this evaluation is a set of new algorithms - valuable in their own right - specifically tailored to these problem classes.

Partial results of this work have appeared in the following publications:

[1] Stuart Bain, John Thornton, and Abdul Sattar (2004) Evolving algorithms for constraint satisfaction. In *Proc. of the 2004 Congress on Evolutionary Computation*, pages 265–272.

[2] Stuart Bain, John Thornton, and Abdul Sattar (2004) Methods of automatic algorithm generation. In *Proc. of the 9th Pacific Rim Conference on AI*, pages 144–153.

[3] Stuart Bain, John Thornton, and Abdul Sattar. (2005) A comparison of evolutionary methods for the discovery of local search heuristics. In *Australian Conference on Artificial Intelligence: AI'05*, pages 1068–1074.

[4] Stuart Bain, John Thornton, and Abdul Sattar (2005) Evolving variable-ordering heuristics for constrained optimisation. In *Principles and Practice of Constraint Programming: CP'05*, pages 732–736.

# Acknowledgements

Portions of this thesis were written whilst on research detachment to the Cork Constraint Computation Centre, University College Cork, Ireland. I am grateful to Professor Freuder for affording me the opportunity and the support to work within his research group.

I am indebted to my doctoral brethren, Owen, Valnir, Lingzhong and Nghia, for their ongoing support and collaboration, and also to Pushkar Piggott for initially directing me towards genetic programming.

I am particularly grateful to my thesis proof readers, Owen Bourne and my mother Lyn Bain, for their efforts.

Without the influence and support of my supervisor Professor Abdul Sattar, this thesis would not have come to fruition. Abdul, I am indebted to you for encouraging me to undertake a PhD in Computer Science. Even before I became one of your doctoral students you gave freely of your support and guidance, and I am grateful for it.

And to my supervisor Dr John Thornton, who devoted so much of his time to me during the course of my studies. John, I believe a quote from you-will-know-who is appropriate at this time: "Men of great intellectual worth, or, still more, men of genius, can have only very few friends." I appreciate being one of them.

# Contents

# Chapter 1

# Introduction

## 1.1 Motivation

Problems involving constraints are ubiquitous in both theoretical and applied computer science. As many real-world problems can be easily formulated as constraint satisfaction problems or constrained optimisation problems, the development of algorithms for solving such problems remains an important area of research. Consider however, the difficulty in choosing from amongst many algorithms, the one most appropriate for a particular class of constraint problems.

That this difficulty exists has been known for some time: the necessity of using different algorithms, for different problems, in order to maximise performance was dubbed the *algorithm selection problem* by Rice (Rice, 1976). Rice presented abstract models to describe the algorithm selection problem but provided no theoretical foundation as to why the algorithm selection problem exists, assuming the above-mentioned necessity instead. It was almost 20 years until the theoretical foundations appeared in the guise of the *no free lunch theorems* (Wolpert and Macready, 1995).

From the emphasis placed upon benchmark performance results in the constraint satisfaction community, it might be supposed that algorithms with the best benchmark performance will be best suited for other problems as well; specifically, those problems of interest to an end-user. According to the no free lunch theorems, this is not necessarily the case. Unless an algorithm exploits particular features of the benchmark problems and these features are also present in the problems of interest, then there is no reason to believe that the algorithm will offer performance superior to that of a randomised algorithm.

This statement must be qualified by the fact that the no free lunch theorems apply only to black-box search algorithms, that is, algorithms that do not exploit knowledge of

1

the underlying structure of the problem on which they are being run. Such algorithms instead rely solely on information about the cost surface of the problem, obtained during a traversal through the search space. Although this qualification makes the theorems inapplicable to some constraint algorithms (Wolpert and Macready explicitly exclude branch and bound search, as it relies on the cost structure of partial solutions), this inapplicability does not mean that the performance of such algorithms can be predicted from their benchmark performance. Furthermore, many constraint algorithms are bound by the implications of the no free lunch theorems. As an example, hill-climbing local search algorithms, like the well-known GSAT (Selman et al., 1992) algorithm.

Consequently, to overcome the difficulties associated with matching algorithms to problems, and to circumvent the limitations imposed on the generality of algorithms by the no free lunch theorems, methods of *adaptive problem solving* (Gratch and Chien, 1996) or *adaptive constraint satisfaction* (Borrett et al., 1996) have been developed. Such methods seek to adapt to the peculiarities of the individual problems with which they are faced, for two reasons. Primarily, it is to achieve performance superior to that which could be achieved by a general purpose algorithm. Secondly, these methods also serve to eliminate (or at least reduce) the involvment of a human expert in the often tedious activity of matching algorithms to problems. This would otherwise be necessary to achieve such superior performance, and has been recognised as a limitation within the constraint programming community (Puget, 2004).

It is these two points that motivate the current work.

## 1.2 Contributions

The intention of this work is to advance the state of the art in adaptive constraint satisfaction[1], for the motivating reasons outlined in the preceding section. To this end, the following contributions are presented:

1. **The identification of five important features of an adaptive system:** The field of adaptive algorithms, particularly with respect to their application to constraint satisfaction problems, has yet to be thoroughly examined. Whilst a number of authors have developed adaptive systems, there has been only limited discussion as to what

---

[1]The phrase *adaptive constraint satisfaction*, rather than the more general *adaptive problem solving*, will be used throughout this work, as it is the more appropriate for a work concerned with the application of adaptive methods entirely to constraint satisfaction problems. It should not be read to imply a greater similarity to the work of Borrett et al. or to discount the relevance of Gratch & Chien's work.

features an adaptive constraint system could exhibit (and perhaps should). This limited discussion may explain why previous implementations have exhibited only subsets of the five important features identified in this work as crucial to the utility of an adaptive constraint satisfaction system.

2. **The development of a three part representation for constraint algorithms:** By analysing the structure of existing constraint algorithms, structures common to both complete and local search algorithms are identified. By explicating these structures, it becomes possible to define constraint algorithms using a three-part representation. These parts are termed *contention*, *preference* and *selection* respectively. The representation will be shown sufficient to represent many of the algorithms currently used in constraint satisfaction, permitting an adaptive system to create algorithms of similar complexity and power.

3. **The development of an adaptive constraint system using genetic programming:** Having identified five crucial features of an adaptive system, it becomes possible to implement a system that exhibits those features. Although some of these features are determined by the choice of an appropriate representation, equally important is the selection of a method of adaptation. Genetic programming aptly fulfills the remaining requirements. This is believed to be the first use of pure genetic programming for learning constraint algorithms.

4. **An evaluation of learning new algorithms for hard, real-world problem instances:** To demonstrate the efficacy of a genetic programming-based adaptive constraint system, a thorough evaluation of the system is provided. The evaluation uses problems that are widely recognised to be both hard and representative of real-world problem instances. Specifically, these instances include randomly-generated problems drawn from known hard distributions, SAT-translated cryptographic problems as well as over-constrained problem instances. The outcome of this evaluation is a set of new algorithms - valuable in their own right - specifically tailored to these problem classes.

## 1.3 Outline

Chapter 2 introduces the domain of constraint satisfaction and over-constrained problems. Formal definitions of these and other relevant terms are presented before a review of the more well-known constraint algorithms, from both the local and complete search domains. This

chapter introduces a three-part representation for constraint algorithms that is appropriate for both local and complete search routines.

Chapter 3 provides an in depth analysis of existing methods for the automatic adaptation of constraint algorithms. The representation described in Chapter 2 is contrasted with some of the representations that have been proposed to operate with adaptive constraint systems. From this analysis, five features important to the efficacy of an adaptive system are identified.

Chapter 4 introduces a system for evolving algorithms for constraint satisfaction, using the representation to be described in Chapter 2. This system uses genetic programming as the method of adaptation, which is discussed in addition to its forebear, the genetic algorithm. The system is shown to meet all five of the important features identified in Chapter 3.

Results of the initial experiments with the framework are presented in Chapter 5, followed by a rigorous evaluation of this system on a range of both benchmark and real-world problems in Chapter 6. The results are discussed, conclusions drawn and potential directions for future work is described.

# Chapter 2

# Representing Constraint Algorithms

This chapter provides the background to constrained and over-constrained problems, and surveys the range of algorithms available to address such problems. This analysis leads to the development of a new, three-part representation for constraint algorithms.

## 2.1 Constraint Satisfaction and Over-Constrained Problems

At the general level a *constraint satisfaction problem* (CSP) is a formalism used for modelling a diverse range of problems. All CSPs are characterised by the inclusion of a finite set of variables; a set of domain values for each variable; and a set of constraints that are only satisfied by assigning particular domain values to the problem's variables (Mackworth, 1977; Montanari, 1974).

When solving a CSP, the goal can be either to determine whether or not a solution exists (if one does, then the problem is termed *satisfiable*) or to find one or more particular solutions to the problem. A solution to a CSP is an assignment of a domain value to each variable such that all constraints are satisfied. The formal definition of a CSP is presented as Definition 1.

**Definition 1** *A CSP is a 3-tuple $CSP = \langle V, D, C \rangle$, where $V$, $D$ and $C$ are all sets. $V$ is a set of variables, each $D_i \in D$ is a set of possible domain values for variable $V_i$, and each $C_j$ is a relation on a subset $S_j$ of the variables in $V$, that defines the combinations of domain values allowed by the constraint. A solution to a CSP is an assignment $A$, mapping each variable $V_i$ to a single domain value in $D_i$, such that all constraints are satisfied. There may be many valid solutions to a particular problem instance.*

This definition is appropriate for problems where a solution exists or for which it is

sufficient to determine that no solution exists. A constraint problem for which there is no solution $A$ that simultaneously satisfies all constraints is termed an *over-constrained constraint satisfaction problem* (OCSP). In the case of over-constrained problems where it is not enough to merely determine that no satisfying solution exists, additional information is required to determine what constitutes an acceptable 'solution'. Such situations lead to optimisation problems, where the goal may be to find the assignment that violates the fewest constraints; one that violates an acceptable number of constraints; or one that minimises a particular cost function.

Many real-world problems require the use of an OCSP formalism, rather than the CSP formalism, due to the absence of an ideal solution (Jampel et al., 1996). A number of formalisms have been proposed to represent OCSPs, differentiated by: the information they require about the constraints; the methods they employ for comparing solutions; and the choice of a metric in either the problem-space or the solution-space. However, despite their differences, all OCSP formalisms extend the definition of a CSP to provide a richer language for the specification of problems.

One commonly used formalism to represent OCSPs is that of *semirings* (Bistarelli et al., 1995, 1997), which provide a powerful representation for the specification of a range of constraint problems, of which OCSPs are just one example. As the power and flexibility offered by the semiring formalism is more than is required for the over-constrained problems examined in this work, a more appropriate simpler definition for an OCSP is presented here instead. The following definition for OCSPs, Definition 2, subsumes most solution-space[1] OCSP formalisms by providing the flexibility to specify an arbitrary cost function, $F$.

**Definition 2** *An OCSP extends the definition of a CSP to a 5-tuple $OCSP = \langle V, D, C, W, F \rangle$ by associating with each element $C_j \in C$ a weight (or cost) $W_j \in W$ that describes the importance of that constraint in the problem. This weight is considered for all solutions that do not satisfy constraint $C_j$, being combined by the function $F$ into a cost metric for the overall solution. The definitions of V, D and C are unchanged.*

---

[1]The difference between a formalism employing a solution-space metric, in contrast to a problem-space metric, is that the former considers the difference between competing solutions, whereas the latter considers the difference between competing reformulations/relaxations of the problem such that the reformulated problem is satisfiable cf. (Freuder and Wallace, 1992).

## 2.2 Types of Problems

Notwithstanding the general definition of a CSP, some forms of CSPs and OCSPs occur more frequently than others making it worth defining them individually. Presented first is the definition of a *binary CSP*:

**Definition 3** *A binary CSP/OCSP is one where each constraint involves at most two variables:* $\forall_{C_j \in C}, \ |S_j| \le 2$

Another important constraint problem is the *propositional satisfiability problem* or simply *SAT problem*, defined formally as follows:

**Definition 4** *A satisfiability problem (or SAT problem) in conjunctive normal form is a constraint problem where the only available domain values for each variable are the logical values* $\{\top, \bot\}$. *Each constraint (termed a clause) is a disjunction of literals (a variable or its logical negation). An instance is satisfiable if a variable assignment exists that provides each clause with at least one true literal. Instances are sometimes denoted n-SAT, imposing the additional restriction that each clause may have at most n literals.*

**Definition 5** *A maximum satisfiability problem (or MAX-SAT problem) is a SAT problem where the objective is not to locate a solution that simultaneously satisfies all clauses, but a solution that simultaneously satisfies the maximum number of clauses. Generally the maximum number of clauses that can be simultaneously satisfied in a MAX-SAT problem instance is less than the total number of clauses in the instance (i.e. the problem is expressly unsatisfiable).*

Although there are methods to translate between these different problem types (Frisch and Peugniez, 2001; Dechter, 2003), the result is often a substantial increase in problem size, as some problems do have a natural representation. Whilst this may permit the use of more efficient algorithms from the other domain, the efficacy of performing the translation must be settled on a case by case basis.

## 2.3 Background to Incomplete Search

Many search methods are described as *local search* because their method of navigating a search space is to make repeated moves within the local neighbourhood of the current tentative solution to a problem. Although not enforced, the local neighbourhood is generally defined to be the modification of a single variable's value assignment. These methods are

more correctly termed *incomplete search*, as their unsystematic traversal of the search space does not guarantee that they will search its entirety within finite time.

Hoos divided incomplete methods into two categories (Hoos, 1998), depending on whether (or not) the probability of locating a solution converges to 1 given infinite time. Those methods, which given infinite time are guaranteed to locate a solution, are termed *approximately complete*. In other circles, specifically the simulated annealing and evolutionary computation communities, this property is more often referred to as *convergence*. Other methods, which are not guaranteed to converge are alternatively termed by Hoos to be *essentially incomplete*.

The implications of this (for both categories) are as follows: local search procedures have worst-case runtime performance of $\mathcal{O}(\infty)$, meaning that they are not guaranteed to find a solution within finite time; they are unable to establish that an instance has no solution[2]; and they are unable to guarantee the optimality of solutions to over-constrained problems (unless the optimal is known in advance).

Whilst this may appear to limit the usefulness of such approaches, in practice, local search has repeatedly been shown to offer good average-case performance on a range of (generally satisfiable) problem types (Selman et al., 1992; Minton et al., 1992b; Le Berre and Simon, 2005a), which has been attributed to local search's ability to exploit perceived gradients in the search space of a problem (Ginsberg, 1999).

The two main features of a local search are the method used to traverse the search space, and the method used to escape from local minima. Practically the two are often inseperable, with many algorithms not including an independent escape routine, but using a method of traversal that inherently includes some form of escape mechanism. This aside, the two mechanisms will be considered seperately in order that a taxonomy may be imposed upon the overall domain.

### 2.3.1   Traversal Mechanisms

#### Stochastic Methods

Whilst almost all local search methods employ some form of stochastic process, either during initialisation or to escape from local minima, some algorithms rely exclusively on these methods for their traversal of the search space as well.

Random traversal of the search space is perhaps the simplest of all possible traversal strategies, and also one of the earliest. Randomised methods such as the Monte-Carlo

---

[2]This excludes the possibility of performing a local search upon a space of disproofs.

method date to as early as 1946 (Eckhardt, 1987). These methods have retained their importance as many modern traversal methods incorporate stochastic processes, even though modern methods are generally more complex. One difficulty in categorising random methods though, is to distinguish between a random walk as used for traversal, as opposed to an explicit trap escape mechanism, as a randomised traversal strategy may implicitly allow a search to escape from local minima.

In the context of a constraint satisfaction problem, a *random walk* refers to the process of making successive random modifications to an (infeasible) solution for the problem. Simple as it may be, a procedure using only a random walk is approximately complete (Hoos and Stützle, 2005) but despite this the procedure is rarely (if ever) used independently. Papadimitriou used a simple randomised method to find solutions for satisfiable 2-SAT formulas (Papadimitriou, 1991). A random walk was also suggested as a possible extension to the greedy descent mechanism of the GSAT (Selman et al., 1992) algorithm, but was described as a trap escape mechanism and not as a traversal mechanism (Selman and Kautz, 1993). However, as it was applied probabilistically throughout the search and not only upon recognising that the search procedure had reached a minima, it should be regarded as method of traversal.

An older method than these is that of *simulated annealing* (Kirkpatrick et al., 1983), originally proposed for combinatorial optimisation rather than decision problems. Inspired by processes from statistical mechanics, simulated annealing emulates the motion of atoms as they are heated and then move towards a low energy state. Simulated annealing operates by randomly sampling the neighbourhood of the solution vector, and calculating whether the point sampled has a lower cost (analogous to lower energy in statistical mechanics) than the current solution. Points that are improvements over the current solution are accepted automatically. This process is not a random walk, but is instead a form of *first descent*, and is not on its own sufficient to make the algorithm probabilistically complete.

The random walk component of simulated annealing is used when considering transitions to points that are worse than the current solution (analagously, a move to a higher energy state). The probability of accepting a move with a cost $\Delta$ higher than the current solution is determined both by this cost differential and by the current *temperature* of the system, $T$, and usually according to a variant of the following equation: $P(\Delta) = exp(-\Delta/k_B T)$. $k_B$ in this equation denotes Boltzmann's constant. Initially, a high temperature frequently allows worsening moves, but as it is slowly reduced worsening moves become increasingly unlikely.

This facility to make random cost-worsening moves is sufficient to make the procedure

approximately complete (Hajek, 1988), or more specifically, to guarantee the convergence[3] of the algorithm. The performance of simulated annealing for solving propositional satisfiability formulas was comparable to that of other routines from around the same time (Spears, 1996; Beringer et al., 1994). Spears makes the suggestion that as problem size increases, simulated annealing scales better than some other algorithms. However the drastic advances made in algorithms descended from GSAT over the last decade, which are discussed in the following sections, mean that simulated annealing is likely no longer competitive on such problems.

Another stochastic method inspired by physics is Survey Propagation (Braunstein and Zecchina, 2004). It requires that the SAT (or MAX-SAT) instance be reformulated as a *factor graph*, which is an undirected bipartite graph with two types of nodes. There is one *variable node* for each variable in the original instance and one *factor node* for each clause in the original instance. Edges in the factor graph only ever connect nodes of different types, ensuring that the graph will be bipartite. There is one edge for every literal in the original instance, representing that the variable is present in that particular clause.

This representation is then used by what the authors describe as an enumerative graph growing algorithm, which determines the set of truth assignments of the original problem. In every iteration, the graph is grown by adding another variable after which any contradictory configurations are filtered out. Although the procedure can operate without a heuristic, it is more efficient to allow the procedure to be guided by the joint probability distribution of a variable's neighbours. Calculation of the joint probability distributions is possible because these distributions are weakly correlated, caused by the "tree-like" structure of the reformulated factor graph.

Although there are some pathological instances where the survey propagation algorithm can be misled, in general its behaviour is comparable to WalkSAT.

**Descent Mechanisms**

Descent mechanisms have already been alluded to in the proceeding section, in part because they too are a rather simplistic (but often quite effective) method for traversing a search space that has been used in many algorithms. The assumption underlying these methods is that infeasible solutions that violate fewer constraints are likely closer to a feasible (or optimal) solution.

The primary problem with descent-based traversal is that the search lacks any form of

---

[3]Although there are similarities between the properties of *PAC* and *convergence*, in many cases convergence results for individual algorithms are often stronger than the more general notion of approximate completeness (Hoos and Stützle, 2005).

foresight, refusing to take any disimproving steps regardless of the potential future benefit. Except on the easiest of problems and the most fortuitous starting points, any descent-based local search algorithm is going to encounter a local minimum at some point. Regardless of how close to the global minimum the search begins, it will be unable to reach it unless there is a continuous sequence of improving moves to it from the starting point.

One of the most common methods is *steepest descent* or *greedy descent*, named for its behaviour of selecting the move from the neighbourhood that offers the best overall improvement in solution cost. Definitions of greedy descent (and other variations as well) generally consider only improving moves, and do not consider equal cost or *plateau* moves as potential candidates, even if no improving moves exist (Bohlin, 2002; Hoos and Stützle, 2005).  In practice however, such plateau moves are considered important and often allowed, as they were in GSAT (Selman et al., 1992), which is one of the best known implementations of a greedy descent procedure. GSAT was specifically proposed for solving propositional satisfiability formulas, however its architecture is easily adapted for generalised CSPs as well. Just as some stochastic methods (simulated annealing, for example) incorporate descent mechanisms, algorithms that are primarily descent-based may be augmented with stochastic features.  An example of this is the GWSAT algorithm (Selman et al., 1994), which introduces a random walk feature into GSAT.

Descent-based search does not include an implicit mechanism that allows it to escape from local minima, and such approaches therefore require the addition of an escape mechanism if they are not to effectively terminate at the first local minima or plateau encountered. It is for this reason that descent mechanisms are not, on their own, approximately complete. Whilst the problem of the procedure becoming stuck can to some extent be mitigated by allowing it to consider plateau moves, this is still insufficient to provide approximate completeness.

One variation of the greedy descent routine is *random descent* (Bohlin, 2002), which selects the candidate move to enact with a probability proportional to the reduction in cost that it offers[4]. Although irrelevant as far as approximate completeness is concerned, the non-determinism of such an approach can prevent the search from becoming trapped within a cycle that can result from some of the more deterministic trap-avoidance mechanisms.

Another common variation is to more efficiently examine the neighbourhood of the current solution. Instead of selecting the lowest-cost neighbouring solution, *first descent* examines neighbours only until one is discovered that offers some improvement, and this move is enacted immediately.

---

[4]Random descent should not be confused with the *randomised iterative improvement* described in (Hoos and Stützle, 2005), which is a descent mechanism used in conjunction with a random walk.

Another descent method that employs a reduced neighbourhood is the *min-conflicts* heuristic (Minton et al., 1992a,b), upon which GSAT was originally based. In contrast to GSAT however, the min-conflicts heuristic is more directly applicable to multi-valued (non-Boolean) constraint satisfaction problems. A single variable is selected at random from amongst those involved in currently violated constraints. All possible domain values for the selected variable are examined before it is greedily reinstantiated with the domain value that will minimise the number of constraint violations.

**Constraint-Directed Methods**

The heuristics so far considered are essentially variable-directed, with their effect on the constraints in the problem considered in aggregate. An alternative is to consider a particular constraint or constraints, and enact moves that ensure the satisfaction of the desired constraint(s), with less regard for the overall effect on the rest of the solution. In the case of local search, constraint-directed methods still entail modifying the variables of an infeasible solution.

Constraint-directed methods exhibit similar behaviours to their variable-directed counterparts. In some respects then, the primary distinction of such methods is that they reduce the size of the neighbourhood that must be examined to the variables involved in just one or some of the constraints of the problem. In the case of large problems, with perhaps tens of thousands of variables, this can be of great practical benefit.

Another distinction is that the variable-directed methods discussed previously, although perhaps designed for either SAT or CSP, are easily generalised to admit heuristics for the other problem type as well. This is not necessarily the case for constraint-directed methods. Any unsatisfied constraint in a SAT problem may be satisfied by a modification to just one of its variables, irrespective of the value of its other variables. This is not the case for a CSP, where in the worst case, in order to satisfy a constraint within the context of the current infeasible solution, it may be necessary to modify all variables associated it. Potentially, this entails that a number of moves exponential in the arity of the constraint must be considered, as opposed to a linear number in the case of SAT. If the ostensible purpose of such methods is, as we suggest above, to reduce the size of the neighbourhood that must be examined, then such methods are of less practical utility for solving generalised CSPs.

WSAT (or WalkSAT) (Selman et al., 1994) and its many valued CSP equivalent NB-WalkSAT (Frisch and Peugniez, 2001) are constraint-directed variants of the GSAT algorithm, which incorporate both stochastic and descent based features. They randomly select

a constraint to satisfy and then choose a variable to modify from that constraint that violates the fewest other constraints (greedy descent). This best move is taken with probability $1 - p$, otherwise a variable from the same constraint is randomly selected to be modified instead (stochastic).

The constraint directed WalkSAT algorithm was the origin of a large family of similar algorithms, beginning with Novelty (McAllester et al., 1997). Like WalkSAT, Novelty is a constraint-directed algorithm, but includes an additional decision step. Novelty records how long it is since each variable has been modified, and will only choose greedily from amongst the variables in a constraint if the would-be selected variable is not also the most recently modified variable of this constraint. If the best variable is also the one modified most recently, then Novelty chooses randomly, but only between the best and the second best variables of the constraint. This is in contrast to WalkSAT, which can make a fully random choice from amongst all of the variables of a constraint. R-Novelty (McAllester et al., 1997) is an extension to Novelty that takes account of the magnitude of the difference between the best and second best variables of a constraint. If the number of constraints that would be satisfied by these variables differs by only one, then a different probability value is used for selecting between the best and second best variables than if they differ by a greater amount. As Novelty and R-Novelty are only choosing between the best and second best variables, they are essentially incomplete (Hoos, 1998).

To address this, Novelty and R-Novelty were augmented with a random walk feature (Hoos, 1998), creating the Novelty+ and R-Novelty+ algorithms. These algorithms differ from their precursors only in that they select (with probability $wp$) a variable that is part of a currently unsatisfied clause. The variable indicated by the respective heuristic is therefore selected only with probability $1 - wp$.

## 2.3.2 Minima Escape Mechanisms

Methods like stochastic traversal are less susceptible to being caught in a local minima than others methods such as descent. As this also means that random methods can be shown to be approximately complete, this might seem to theoretically favour the use of such methods. Practically though, purely random methods are rarely used in practice, as they are inefficient compared to heuristically guided methods. However, methods that are susceptible to becoming stuck in local minima require some form of escape mechanism if they are to be expected to solve any but the simplest problems.

**Stochastic Methods**

The simplest way to escape from such a position is simply to *randomly restart*, reinstantiating every variable with a random value in the hope that this time the search will begin closer (or with a more direct path) to the global minimum. Such a strategy is inefficient however, since no information about the search is retained from restart to restart, and although the search may previously have come very close to achieving the goal, any knowledge of that achievement is lost with the next restart.

A less drastic approach could be described as *random perturbation*. This involves making one or several random moves in an attempt to escape from a local minima. Since the values of most of the variables are preserved, the search effort expended in order to discover them is not wasted. Unfortunately, it depends on the topography of the search space whether such moves will be sufficient for the search to escape the local minimum, or whether it will immediately fall back into the trap.

**Memory Methods**

To prevent a search revisiting the same positions in the search space, most importantly the same local minima, some searches maintain a memory of where they have been. This allows the search to safely make cost-worsening moves as it protects the move from immediate retraction for as long as the memory persists.

One of the most well-known memory methods is *tabu search* (Glover, 1989, 1990). Ideally, tabu search would be implemented by recording each point in the space that the search passed through on a tabu list, with states on the tabu list never to be revisited. Memory considerations aside, if visited states were never 'forgotten', an otherwise incomplete local search procedure would become a complete systematic search procedure. Whilst Glover's description of cancellation sequence would allow the realisation of such a system, practical implementations of tabu lists are generally much simpler and of a fixed length, meaning that entries persist only for so long, and record only the variable modified and not the full state description. Under such an implementation, the search procedure is not permitted to modify the value assignment of any variable on the tabu list.

By disallowing any modifications to a tabu variable, many potential solutions (most of which have probably not, in fact, been visited) become disallowed. To overcome the fact that some good solutions may have inadvertently become tabu, an *aspiration* condition is introduced, permitting modifications to a tabu variable, if they would lead to a solution with a cost better than has previously been observed. The rationale behind this is, that since the

solution under consideration has a cost better than any the search has so far observed, it must never have been visited previously. Moving to such a solution state therefore doesn't violate the requirement that tabu search visit a solution no more than once.

Whilst tabu search is a meta-heuristic and can be used with a wide range of other search heuristics, it also allows procedures that exploit the fact that they will not return to a previously visited local minima. One example of this is an algorithm specifically for over-constrained MAX-SAT problems, the *steepest ascent mildest descent* heuristic (Hansen and Jaumard, 1990). This particular heuristic considers only those variables flipped in non-improving steps to be tabu, but also moves more predictably away from local minima than do stochastic trap escape mechanisms.

**Hybrid Methods**

The fact that an ideal implementation of tabu search would convert an incomplete search procedure into a complete search was discussed in the preceding section. A different method that would ultimately achieve the same end is that of *horizon extension* (Bohlin, 2002). Beginning with a small neighbourhood, such as the modification of just one variable, this method involves incrementally increasing the size of the local neighbourhood under consideration to the extent that the search can see beyond the local minima and move out of it.

Whereas horizon extension implicitly leads towards complete search procedures, other hybrid methods more explicitly combine complete search with local search routines. Dynamic backtracking was combined with GSAT (Ginsberg and McAllester, 1994), allowing a greedy local search to escape local minima through dynamic backtracking's recording of known infeasible solutions (no-goods). Alternatively, the behaviour of a hybrid method can be varied between the two extremes of complete and incomplete search, as described in (Zhang and Zhang, 1996), where a parameter of the search procedure is used to vary the operational behaviour of the search. The parameter of their algorithm controls the depth at which the search begins by altering the number of variables of a problem that are initially instantiated. Instantiating fewer variables causes the algorithm to behave more like a complete search routine, whereas a greater number of initially instantiated variables modifies the algorithm's behaviour to emulate local search behaviour.

**Perturbation**

All of the algorithms discussed so far have operated on search spaces that are dependent only on the specification of the problem. Local minima were accepted as features of the search space that were to be escaped, rather than as features that might be eliminated. Perturbation methods perturb the original cost surface of the problem in an attempt to eliminate any local minima that they become trapped in. By extension, eliminating a minimum means that the procedure is no longer trapped by it.

Over time, these methods have been known by a number of different names: clause (constraint) weighting (Selman and Kautz, 1993), after their behaviour of adding weights to constraints; Lagrangian methods (Wah and Shang, 1997), after the Lagrange multipliers used in continuous optimisation; dynamic local search (Hoos and Stützle, 2005), as the search space operated on is modified dynamically; and also non-Markovian methods (Anbulagan et al., 2005), as future moves are not independent of past ones. At present, dynamic local search appears to be the most popular name for such methods.

These methods originated from the observation that greedy search methods, specifically GSAT, encountered difficulties solving problems with asymmetric structure. This prompted Selman and Kautz to propose a strategy for varying the importance of certain constraints (Selman and Kautz, 1993), and also for Morris to independently propose the Breakout method (Morris, 1993). In both methods, any constraints remaining unsatisfied after a number of iterations have their weight (violation cost) increased. For example, satisfying a constraint with a weight of two is equivalent to satisfying any two ordinary, unweighted constraints. In this way, constraints that are consistently difficult to satisfy accrue sufficient weight that they either remain satisfied, or direct the algorithm to another part of the search space where they are easier to satisfy.

Adding constraint weights was likened to duplicating constraints in (Selman and Kautz, 1993), but an alternative to this direct duplication was proposed in (Cha and Iwama, 1997) by Cha and Iwama. Their proposition was to generate new constraints, resolving each unsatisfied constraint with its neighbouring constraints (neighbouring constraints are ones that share a common variable with the unsatisfied constraint but require a different value for that variable). As presented, their work is only applicable to the satisfiability problem, as satisfying the new resolved constraint would also satisfy the two original constraints.

With the advent of weighting methods, it was quickly realised that monotonically increasing the weight on constraints (or features) would reduce the effectiveness of future weight increases (Frank, 1997). This would negatively affect performance as more weight

increases would be required before there would be any noticeable effect on the search space. This can be overcome by either continually or periodically reducing the weight on certain constraints, generally once they have become satisfied. Beginning with Frank's work (Frank, 1997), similar weight reduction heuristics now compose part of all the major weighting algorithms, including those that multiplicatively (rather than additively) alter the weights on constraints.

Some early weighting algorithms added weight only after a given number of moves, making these methods unable to advance significantly until the end of that period. Frank proposed a variation (Frank, 1996) of Weighted GSAT where the weights on unsatisfied constraints are modified after every move. This method provides feedback to the algorithm much more quickly than in the previous work and proved to be more effective. However, as adding weights is computationally expensive, a reasonable extension was to add weight significantly less frequently, such as only when search encounters a local minima, as is done in both Breakout (Morris, 1993) and the discrete Lagrangian method, or DLM (Shang and Wah, 1998; Wah and Shang, 1997).

Lagrangian methods of optimisation have traditionally been applied to problems involving continuous variables, rather than the discrete variables often considered in constraint satisfaction. In order to use such methods to solve CSPs it is necessary to provide analogous discrete domain definitions of gradient and the Lagrangian function (Shang and Wah, 1998; Wah and Shang, 1997). These theoretical results led to the development of DLM. When DLM reaches a local minimum in the search space, instead of restarting from a new location that may not be any closer to the overall solution, adjustments to the Lagrangian multipliers effectively eliminate the local minimum from the algorithm's objective function. When applied to SAT, DLM can be considered equivalent to the earlier constraint weighting algorithm of (Morris, 1993). However, a tabu list and the ability to reduce the Lagrangian multipliers on constraints was introduced in (Shang and Wah, 1998) to control the search on plateau areas and make the algorithm more competitive.

Guided Local Search (GLS) (Voudouris, 1997; Mills and Tsang, 2000) is similar to DLM in that it modifies the search space only upon encountering a local minimum. GLS is a meta-heuristic capable of recognising undesirable 'features' of a local minimum and penalising these features to force the search to move elsewhere. The algorithm is not restricted to any particular type of search problem, but requires the user to select a feature of the problem under consideration to be penalised upon encountering local minima. For constraint satisfaction problems, the features identified in (Mills and Tsang, 2000) were the unsatisfied constraints of the current solution. In this way, the search learns to prefer solutions that

satisfy the penalised constraints, which in practice are the ones that have been more difficult to satisfy in the past. As with DLM, the later extensions to GLS allowed for the penalties attached to constraints to be decreased.

Smoothed Descent and Flood (SDF) (Schuurmans and Southey, 2001) is a greedy descent method (specifically for the satisfiability problem) that multiplicatively re-weights unsatisfied constraints, effectively 'flooding' local minima. However, the novelty of this approach is that it doesn't just recognise that a constraint is satisfied, but takes account of the degree of satisfaction of each constraint[5]. SDF only reduces weights on satisfied constraints, by a process of 'smoothing'. The Exponentiated Sub-Gradient algorithm (ESG) (Schuurmans et al., 2001) evolved from SDF and differs from its progenitor in a number of ways. ESG employs a constant multiplicative update of constraint weights; a regular smoothing (reduction) of all constraint weights (as opposed to satisfied constraint weights in SDF), and a modified penalty function capable of rewarding satisfied constraints as well as penalising unsatisfied ones. Like SDF before it, despite being competitive in terms of the number of moves, the overall time required by ESG was not significantly smaller (and often exceeded) the time required by DLM. This is most likely the result of the added computational overhead of the multiplicative weight updates (as opposed to the additive weight updates of DLM).

Closely related to the ESG algorithm are the SAPS and RSAPS algorithms (Hutter et al., 2002). To increase the speed of search, SAPS applies the computationally expensive weight smoothing operation solely to unsatisfied constraints and less frequently than ESG. Although the probabilistic smoothing used in SAPS resulted in the introduction of a new parameter, it is possible for the search to automatically determine this parameter, as occurs in the 'reactive' (RSAPS) variant.

Although for a time multiplicative methods were at the fore, the benefit of using simpler additive methods was re-established, resulting in the development of the pure additive weighting scheme (PAWS) (Thornton et al., 2004). PAWS would add weight only in local minima, and would subtractively decrease the weight on all weighted constraints after a number of increases determined by its sole parameter. PAWS was shown to offer superior performance than DLM, and with only one parameter also significantly easier to tune than SAPS (with three parameters) or even RSAPS (with two).

---

[5]For the satisfiability problem, this means that solutions in which more than one literal is satisfied in a clause are preferred. This preference takes the form of an exponentially diminishing sum for each additional satisfying literal.

## 2.4 Background to Complete Search

In contrast to local search routines, methods of *complete search* systematically examine a search space in its entirety, bypassing only those areas that are known not to contain a solution. This thorough exploration guarantees that complete methods will find an (optimal) solution if one exists. This, along with the ability to prove a problem unsolvable, is the main advantage offered by complete search. As systematic methods consider each solution at most once, they have been described as efficient (Pearl, 1984), and are therefore not susceptible to repeatedly exploring the same area, as local search may.

The main disadvantage of complete methods is that as CSPs are in general NP-complete, the time-complexity of these algorithms is exponential in the worst case. This means that many large and real-world problem instances are beyond the reach of complete solvers. Even though their worst-case performance is strictly better than that of local search, complete search algorithms are often outperformed by local search in terms of average case performance (Selman et al., 1992). Furthermore, although complete search can establish that an instance has no solution whereas local search cannot, for all practical purposes the search space of large unsatisfiable problems is often too large for them to be handled by complete methods (Ginsberg, 1999).

### 2.4.1 Chronological Methods

Chronological methods are the simplest form of complete search for solving constraint problems. Variables are instantiated in a static, arbitrary order, either according to the problem specification or perhaps at random. Because the ordering is static, each level of the search tree corresponds to one particular variable. Although chronological methods would rarely be used in practice, they are discussed here to introduce three important algorithms that form the basis of dynamic and more advanced heuristic complete search routines.

**Generate and Test**

The simplest (but also least efficient) complete search routine to solve CSPs is *generate and test*[6]. No one work is uniformly cited as the origin of the method, but it is widely recognised within the AI community, with the phrase "generate and test" appearing in text books in the late 70s and early 80s (Winston, 1984). An earlier work (Golomb and Baumert,

---

[6]The phrase *generate and test* is used in this context to refer only to systematic search procedures, and is distinguished from unsystematic sampling from a solution space, which is alternatively termed uninformed random picking (Hoos and Stützle, 2005)

1965) termed this method "the brute force approach". This method operates by sequentially assigning a value to each variable until all have been instantiated. This variable assignment defines a potential solution, which is then tested against the constraints to determine if they are all satisfied, in which case the procedure will terminate. With respect to solving OCSPs, generate and test must store the lowest cost solution that it encounters. If constraint violations are present, the most recently assigned variable that still has domain values left to try is instantiated with a different value. Any subsequent variables (those occurring later in the variable ordering) must have all of their domain values re-examined. This procedure continues until either a solution is found or until no variable has any domain values remaining untried.

**Backtracking**

*Backtracking* is closley related to generate and test, and still performs a systematic search of the solution space but prunes unsuccessful branches from the search tree to improve performance. In contrast to generate and test, which only examines complete solutions, backtracking tests partial solutions (that is, solutions where some but not all of the variables have been assigned a value) against each constraint as soon as all variables relevant to that constraint are instantiated. Contradictions are therefore identified higher in the search tree, allowing the subtree below the violating node to be *pruned*. Pruning the search tree bypasses the exploration of areas that are known not to contain a solution, significantly reducing the overall time and number of constraint checks performed by the algorithm. Apart from the checking of partial solutions, backtracking and generate and test are otherwise identical, each using the same method of variable reinstantiation.

For use in solving OCSPs, a modified version of the backtracking procedure called *branch and bound* is used. The difference between it and a standard backtracking search is that it can only backtrack once the number of constraint violations exceeds those in the best solution found so far (the bound). Unless a known cost bound is specified by the user, such a procedure will likely have to travel deep into the search tree to locate low cost bounds, and must explore as far as the leaf nodes at least once. Although the performance of a branch and bound search is not comparable with that of backtracking, the fact remains that branch and bound search must overlook violations that would have permitted pruning in a corresponding backtracking search.

The earliest use of backtracking in computer science was in Davis and Putnam's work on proving quantified logic statements (Davis and Putnam, 1960). Their algorithm was

specifically designed for satisfiability problems, and contained an additional step to the backtracking procedure described above. This step is called *unit propagation*, and exploits the nature of the satisfiability problem, which requires that any solution provide at least one true literal in every clause. As soon as an unsatisfied clause has only one uninstantiated literal remaining, unit propagation dictates that this literal must evaluate true and form part of the solution.

If recognising that a literal must evaluate true is instead interpreted as recognising that it must not evaluate false, then unit propagation can be viewed as analogous to enforcing arc-consistency in a CSP. In the case of SAT translations of binary CSPs, the type of consistency enforcement effected by unit-propagation will depend on the particular encoding used (Gent, 2002).

**Backjumping**

Constraints can occur between variables widely seperated in the variable ordering. In situations where the instantiation of a variable early in the search order ($V_i$) causes an inconsistency with a variable much later in the search order ($V_{i+j}$), the backtracking search procedure is destined to *thrash* (Bobrow and Raphael, 1974). Thrashing is the rediscovery of the same inconsistency between variables $V_i$ and $V_{i+j}$ for all other assignments to variables $V_{i+1}$ through $V_{i+j-1}$, before the algorithm finally traverses sufficiently far back up the tree to reinstantiate the culprit variable $V_i$.

*Backjumping* (Gaschnig, 1978) is a procedure that identifies such situations and immediately returns to the culprit variable, preventing thrashing and reducing the number of constraint checks. As each domain value of a variable is examined, if the value is found inconsistent then the deepest variable in the tree causing the inconsistency of this value is recorded. After all domain values for this variable have been examined the algorithm immediately returns to the culprit variable highest in the tree, reinstantiating it. A backjumping algorithm may also be extended to solve OCSPs using a branch and bound search analagous to that described for backtracking (Freuder and Wallace, 1992).

## 2.4.2   Heuristic Ordering Methods

The foregoing discussion of tree search algorithms has so far assumed an arbitrary, and fixed, variable ordering i.e. the order of instantiation being determined at the time the search begins. Although this is one of the easiest ways to implement a tree search algorithm, a bad choice of ordering can significantly degrade performance. The superior performance

of backtracking and backjumping over generate and test is due, in part, to the pruning that these algorithms perform.

However pruning and backtracking are only possible if the variables instantiated so far have disallowed all values for a variable, such that any further progress deeper into the current search tree would be pointless. The deeper that pruning occurs in the tree the more closely the algorithm's performance will approximate generate and test. A bad ordering forces the algorithm to explore deep into the tree before detecting an inconsistency, wasting time and permitting very little to be pruned. Rather than use an arbitrary variable ordering, heuristic ordering methods attempt to maximise their performance by employing some rationale behind their choice of an ordering.

As with the complete methods based on chronological orderings, it is possible to use variable ordering heuristics as part of a branch and bound procedure to solve OCSPs.

**Static Ordering Heuristics**

During search it is possible to instantiate the variables in a fixed order independent of the actual effects of the instantiations that occur as search progresses. Chronological backtracking is an example of this, which uses the most trivial of all possible heuristics: the lexicographic position of a variable in the description of the problem. Whilst there are many other more complicated heuristics that might be used, such methods have attracted little attention because dynamic ordering heuristics, which give a greater degree of control, are generally able to outperform their static counterparts (Purdom, 1983).

Static orderings may simply order variables using recognised measures from the problem's constraint graph, such as variable degree (the total number of variables that share a constraint with this variable) or the size of the constraint set (the number of constraints that this variable participates in). More complicated orderings can be constructed, which can also incorporate measures that would generally only be used in dynamic heuristics, i.e. measures that involve the instantiation of variables. In such cases, a valueless 'instantiation' to each variable is considered. For example, it is possible to construct a static backward degree heuristic by choosing during each step of the order construction the variable with the greatest number of 'instantiated' neighbours. Ties (such as for the initial variable choice) may be broken either randomly, lexicographically or by some other measure.

**Dynamic Ordering Methods**

With a static ordering, two different subtrees rooted at the same level are explored in an equivalent way. The alternative to this is a dynamic variable ordering, where the decision as to which variable to instantiate next depends on the effects of previous instantiations. The variable considered for a subtree at one level can differ entirely from the variable selected in another subtree at the same level. This is also the case if the algorithm reaches a similar point, where the same variables have been instantiated but with different values, i.e. the algorithm may make a different decision as to which variable to instantiate next. The goal of this is to ensure that at any time, the most promising variable is instantiated next, although the definition of "promising" depends on the choice of heuristic. Whilst there is a computational cost associated with applying the heuristic at each level in the search tree, the demonstrated superior performance of dynamic methods (Purdom, 1983) is evidence that the cost is justified.

Many different dynamic heuristics have been proposed, often with a particular problem type in mind, even though, with some modification, many might be applied more widely. Such modifications will not be discussed here, and heuristics will be described with respect to the problem that they were first presented for.

Heuristics need not operate by guiding the search towards a solution, but can instead seek out failure. The rationale behind this is that early failures, that is, those that occur high in the search tree, prune exponentially larger sub-trees than those occurring more deeply in the tree. A number of "fail-first" heuristics were collected in (Smith and Grant, 1998), in which the authors attribute the underlying original concept of attempting to fail early to (Haralick and Elliott, 1980). One method defines the preferred variable to be the one with the fewest domain values remaining. Should the current branch result in an unsatisfiable sub-problem, having fewer domain values to try means that the size of the refutation needed to demonstrate unsatisfiability should (if everything else were equal) be smaller. An extension to this heuristic was to break any initial ties in this heuristic by the variable with the highest degree (Frost and Dechter, 1994), as such an instantiation will influence the greatest number of other variables. Rather than just for breaking initial ties, degree information may be used throughout the search. Selecting the variable that minimises the ratio of domain/degree (Bessière and Régin, 1996) favours variables with small domains and/or large degrees. Other extensions to the concept of "fail-first" heuristics include removing the assumption that everything else is equal, and taking into account the estimated tightness of the constraints between variables; the domain sizes of a potential variable's neighbours;

or the true tightness of the constraints between variables (Smith and Grant, 1998). These varied heuristics were termed FF2, FF3 and FF4 respectively.

The inclusion of a measure of the domain size of a potential variable means that these heuristics are not directly applicable to boolean-valued CSPs, such as the satisfiability problem. There is potential evidence however, that failing first is similarly effective when solving satisfiability problems. Hooker and Vinay proposed the "simplification hypothesis" (broadly stated: choose the variable that simplifies the problem the most) (Hooker and Vinay, 1995) in favour of the earlier "satisfaction hypothesis" (a good variable choice is one that maximises the probability of branching into a satisfiable sub-problem). Their proposition was intended to explain the better performance of simplification heuristics than satisfaction heuristics for the satisfiability problem. As one method of simplifying a problem is to prune large amounts of the problem space by failing early in the search, the superior performance of simplification heuristics supports the view that failing-first is beneficial for satisfiability testing. Heuristics such as the "Maximum Occurrences in Disjunctions of Minimum Size" (MOMS) (Freeman, 1995) and the "Two-Sided Jeroslow Wang" (2SJW) rule of (Hooker and Vinay, 1995) are two heuristics that tend to produce simpler sub-problems. The difference between the two is that MOMS considers only the clauses of minimum size whereas 2SJW calculates a weighted sum of clause lengths, so that short clauses, not just the shortest, play a role in variable selection. The variable selection heuristic of SATZ (Li and Anbulagan, 1997), which remains one of the best satisfiability solvers, explicitly counts the simplification that variables would produce in the problem and chooses the one that would maximise the count. Satisfaction style heuristics include the original Jeroslow-Wang rule (Jeroslow and Wang, 1990), the various approximations to it (Hooker and Vinay, 1995), as well as more simplistic heuristics such as simply ordering literals by the number of clauses that they appear in.

Modern, state-of-the-art solvers generally use more complex branching rules, which can be responsive to the progress of the search rather than just the problem instance. This includes heuristics like the "Variable State Independent Decaying Sum" (VSIDS) heuristic, which records for each variable the number of times the variable has occurred in falsified clauses and caused a backtrack. The VSIDS heuristic is in some ways analagous to the dynamic local search procedures described earlier. It has been implemented in a number of solvers, most notably zChaff (Moskewicz et al., 2001) and MiniSAT (Eén and Sörensen, 2003).

## 2.5  Other Algorithmic Methods

The algorithms discussed in this section are fundamentally different to those presented in the foregoing discussion. They are included here primarily because they are often used in conjunction with other local and complete search procedures.

### 2.5.1  Population Based Methods

**Evolutionary Methods**

A full discussion of genetic search methods is not presented until Chapter 4, but the application of such methods to solving constraint problems is summarised here. Evolutionary methods are a form of local search and so exhibit the same limitations as other incomplete algorithms. However, instead of a single solution these methods (most notably the *genetic algorithm* or *GA* (Holland, 1975)) maintain a population of *chromosomes*: candidate solutions to a problem, which in the case of constraint problems, consist of potential variable-value assignments. In general, each chromosome would have as many *genes* as there are variables in the problem, and each gene would be comprised of sufficient *alleles* to represent all possible domain values for that variable.

During the course of the search, the chromosomes are likely to represent infeasible solutions to the problem, as the procedure would generally terminate upon discovery of a feasible (valid) solution. A heuristic function is used to approximate the relative merit of each candidate solution (its *fitness*), which then determines the likelihood that the particular chromosome will participate in some form of reproduction. Reproduction is facilitated by the *genetic operators*, the most common of which mimic sexual recombination, cloning and mutation. As highly fit elements are more likely to be selected to reproduce, chromosomes exhibiting highly fit substructures tend to proliferate over time.

Although individuals in the population are expected to improve from the implicit operation of the algorithm, it is possible to augment a genetic algorithm with methods that more explicitly improve individuals. Such methods are termed *memetic algorithms* (Moscato, 1989), and operate by applying some other form of search (local or systematic) to optimise individuals independently from the rest of the population. The most reasonable time to apply the memetic step is after the genetic operators have been used to form a new individual, so that the fitness of the optimised individual may be considered during selection.

**Algorithm Portfolios**

An *algorithm portfolio* (Gomes and Selman, 2001) can consist of a collection of different algorithms or of multiple instances of the same stochastic algorithm, in which the randomised behaviour of each instance provides performance variability. Particularly in problem domains that exhibit heavy-tailed behaviour (Gomes et al., 1997), the large run-time variance present in some randomised algorithms can be exploited to obtain better overall performance than could be achieved by a single algorithm.

Although algorithm portfolios are described as embodying a population of algorithms, they may alternatively be seen as encompassing a population of solutions (much like in a genetic algorithm). The crucial difference being that where solutions in a GA population are inter-dependent, the solutions generated from an algorithm portfolio are all independent, and each is incrementally extended only by its own algorithm.

## 2.5.2   Consistency Methods

Forward checking is a look ahead strategy that assists in preventing a search from encountering dead ends by applying arc consistency to tentative variable choices. Arc consistency ensures that every remaining domain value for a variable has a supporting (consistent) value in each other variable with which it shares a constraint. Full look ahead continues to propagate the effects of arc consistency from tentative variable choices until no more values can be eliminated or until an inconsistency occurs. However, arc consistency is usually only employed in a limited fashion (partial look ahead) as the application of full arc consistency is too costly (Dechter and Frost, 1998).

Before instantiating a variable with a tentative value, the effect of instantiating this variable on the domains of other variables is considered. Domain values of other variables that would produce inconsistencies are flagged. If all domain values of another variable become flagged, the tentative assignment would result in an inconsistency and can be overlooked. This means that inconsistencies between variables widely seperated in the variable ordering are detected at the higher variable, rather than the lower one, preventing unnecessary instantiations.

*Maintaining arc consistency* (MAC) (Sabin and Freuder, 1994) is an alternate method of consistency maintenance to forward checking. The primary difference between the two is that when a variable is instantiated, MAC removes all other domain values for that variable and then propagates the effect of this to all neighbour variables. The network is therefore 'maintained' in an arc consistent state.

Consistency methods are more implicit when applied to CSPs involving Boolean variables, as flagging one value of a variable as inconsistent necessarily entails that the opposite value must form part of a solution (or that the current partial solution is not extensible to a full solution). As mentioned briefly earlier, unit propagation implicitly implements a form of consistency maintenance that would normally have to be explicitly performed in the case of a CSP with non-Boolean variables. Considering SAT translations of binary CSPs, for which it is possible to examine the operation of consistency techniques applied to the original problem, different translations effect different methods of consistency maintenance (Gent, 2002). Unit propagation applied to a direct encoded problem[7] will behave as would forward checking applied to the original problem. Applied to a support encoded problem[8] however, unit propagation behaves as does MAC on the original problem.

## 2.6 A Three-Part Representation for Contraint Algorithms

The adaptation of constraint satisfaction algorithms requires a representation that is conducive to adaptation and sufficiently powerful to represent the types of algorithms that would otherwise be developed by human researchers. Having considered a wide range of search procedures, such a representation can be developed. Although generality is desirable, development of the representation will initially consider only its application to complete and incomplete search procedures as delineated, and not to those methods categorised as "other". A discussion of this representation with respect to such methods follows.

Both systematic and local constraint satisfaction algorithms can be viewed as iterative procedures that repeatedly assign domain values to variables, terminating when all constraints are satisfied, the problem is proven unsolvable, or the available computational resources have been exhausted. The traditional difference between these two methods is that backtracking search instantiates variables only up to the point where the instantiation is consistent with the constraints, whereas all variables are instantiated in local search regardless of constraint violations. Put another way, systematic search operates within the domain of feasible partial solutions whereas local search operates on infeasible but total solutions.

Despite these differences, at every iteration both types of search make two decisions: "What variable will be instantiated next?" and "Which value will be assigned to it?". These questions are generally addressed by some form of heuristic assessment of the problem,

---

[7]The direct encoding creates at-most-one and at-least-one clauses for the values of each variable, and binary conflict clauses to disallow inconsistent assignments between variables.

[8]The support encoding omits the conflict clauses of the direct encoding in favour of clauses indicating which values of one variable support a particular consistent assignment to another (Kasif, 1990).

leading the algorithm to make variable-value assignments (in the case of complete search), or reassignments (in the case of local search). Such a variable-value re/assignment will be termed a *move*. Generic backtracking and local search procedures are shown as Algorithm 1 and Algorithm 2 respectively. These algorithms clearly show the similarities in the way the two procedures each select (according to heuristic merit) one *move* out of the complete set of possible *moves*. Other details of an algorithm's operation, such as backtracking, constraint propagation - even the instantiation of the selected move - can be abstracted away as meta-level processes.

---

**Algorithm 1**: `Backtrack`($assignment, CSP$): A generic backtrack procedure

**Input**: A partial (but consistent) *assignment* to a constraint problem, $CSP$.
**Output**: A solution to $CSP$ iff *assignment* can be extended to one, otherwise $\perp$

```
/* Check if a solution has been found */
```
**if** `complete`($assignment, CSP$) **then return** $assignment$;

$Moves \leftarrow$ `variables`($CSP$) $\times$ `values`($CSP$);

```
/* Choose one move out of all possible moves */
```
$Contenders \leftarrow \{m \in Moves \mid$ `unassigned`($m_{var}, assignment$)$\}$;
$Preferences \leftarrow \{$`heuristic`($m$) $\mid m \in Contenders \}$;
$move \leftarrow$ `choose`($Contenders, Preferences$);

```
        /* These are the meta level activities of the algorithm */
```
```
/* For completetry all values for the selected move's variable */
```
$MovesToTry \leftarrow \{m \in Moves \mid m_{var} = move_{var}\}$;
**foreach** $moveToTry \in MovesToTry$ **do**
  $\quad assignment \leftarrow assignment \oplus moveToTry$;
  $\quad$ **if** `consistent`($assignment, CSP$) **then**
  $\quad\quad solution \leftarrow$ `Backtrack`($assignment, CSP$);
  $\quad\quad$ **if** $result \neq \perp$ **then return** $solution$;
  $\quad$ **end**
  $\quad assignment \leftarrow assignment \ominus moveToTry$;
**end**
**return** $\perp$;

---

Viewing the operation of a constraint satisfaction algorithm as a sequence of moves necessitates that an algorithm must at each iteration: (1) determine what the candidate moves are, and then usually (2) a heuristic ranking of candidates, before (3) one specific move is selected to enact. These three distinct stages have been termed *contention*, *preference* and *selection* respectively (Bain et al., 2004a)[9]. Each stage is a functional expression of an appropriate type, composed from *measures* describing the nature of the problem and the state of the search.

Such a generic three-part search procedure is shown as Algorithm 3. Each call to

---

[9]related work

---

**Algorithm 2**: LocalSearch($assignment, CSP$): A generic local search procedure

---

**Input**: A complete (perhaps inconsistent) $assignment$ to a constraint problem, $CSP$.
**Output**: A solution to $CSP$ iff a consistent $assignment$ is found.

```
/* Check if a solution has been found */
```
**if** consistent($assignment, CSP$) **then return** $assignment$;

$Moves \leftarrow$ variables($CSP$) $\times$ values($CSP$);

```
/* Choose one move out of all possible moves */
```
$Contenders \leftarrow \{m \in Moves \mid$ allowable($m_{var}, assignment$)$\}$;
$Preferences \leftarrow \{$heuristic($m$) $\mid m \in Contenders \}$;
$move \leftarrow$ choose($Contenders, Preferences$);

```
        /* These are the meta level activities of the algorithm */
```

```
/* Enact the selected move */
```
$assignment \leftarrow assignment \oplus move$;
**return** LocalSearch($assignment$);

---

`HeuristicSearch` constitutes a single iteration of a search procedure, the three stages of which are defined by the functions `contention`, `preference` and `selection` respectively. Defining an algorithm therefore requires a commitment to a specific heuristic for each of these. `MetaSearch`, as well as incorporating all activities relating to the re/instantiation of the selected move, may make additional calls to `HeuristicSearch`.

---

**Algorithm 3**: HeuristicSearch($assignment, CSP$): A generic search procedure

---

**Input**: A tentative $assignment$ to a constraint problem, $CSP$.
**Output**: A solution to $CSP$ iff a consistent $assignment$ is found, or $\perp$ if the problem is proved unsatisfiable.

```
/* Check if a solution has been found */
```
**if** solves($assignment, CSP$) **then return** $assignment$;

$Moves \leftarrow$ variables($CSP$) $\times$ values($CSP$);

```
/* Choose one move out of all possible moves */
```
$Contenders \leftarrow \{m \in Moves \mid$ contention($m_{var}, assignment$)$\}$;
$Preferences \leftarrow \{$preference($m$) $\mid m \in Contenders \}$;
$move \leftarrow$ selection($Contenders, Preferences$);

```
        /* These are the meta level activities of the algorithm */
```
$result \leftarrow$ MetaSearch($move, assignment, CSP$);
**return** $result$;

---

The three stages of such a search procedure will now be discussed in more detail.

**Contention:** The purpose of contention is to determine the moves available to the algorithm at the present time. As two trivial examples, consider: the most general contention function for complete search is the rule that a move is in contention if its variable is currently uninstantiated and the value not already tried within the current context; or the contention

function for a local search with a tabu list, where a move remains in contention only if its variable has not been recently modifed.

Although it might be possible to represent an algorithm without considering contention, at times it is desirable that some moves not be considered at all (as the above examples attest), irrespective of how heuristically meritorious they may appear. Without contention, this would require the use of arbitrarily large (preference) values to ensure that such moves occur at the bottom of the preference ordering. Even then, some selection functions might still be inclined to select such moves in spite of their low preference ranking. Eliminating the contention stage would also complicate the matter of preference heuristics, which would be required to assign values to some moves related more to their contention than to their preference.

Specifically, a contention heuristic may be any function that potentially disallows a move. One example was that used in the generic backtracking procedure: `unassigned`, which only allows moves relating to variables that are currently uninstantiated. Another is the tabu criteria, commonly seen in local search: a move remains in contention only if it is not found on the tabu list.

A contention heuristic is essentially a mapping *Move → Bool*. The output of the contention stage is a set of moves, being the subset of the universal set of moves for which the contention heuristic evaluates as *True*. Only the moves remaining in contention will have their heuristic merit calculated in the preference phase.

**Preference:** The second stage of each iteration is to assign a numerical preference value to all of the moves that remain in contention. Two examples of preference heuristics are: the number of constraints that would be satisfied if this move were taken; or the number of neighbours that this move's variable has in the constraint graph.

Preference performs a mapping *Move → Numeric* for every move in the contention set. For generality, the output of a preference heuristic is required only to be numeric. It is an implementation matter to decide whether the measures involved more naturally map to the Integers or the Reals.

There are many preference heuristics that may be used to rank candidate moves, some examples of which are: the number of neighbours the move's variable has in the constraint graph (degree); the number of constraints the move or variable participates in; the number of remaining domain values for a move's variable (domain size); et cetera.

Any move-specific information necessary to distinguish between candidates (including secondary tie-breaking heuristics) should be calculated and included as part of the preference value. Equivalent preference values are just that, and indicate that there is no way to further

distinguish between such moves. The easiest way for secondary tie-breaking information to be included as part of the single preference value is to multiply the primary measure by some constant and then add the secondary tie-breaking value to the result. The constant need only be sufficiently large that it always dominates the tie-breaking measure.

**Selection:** Given a preference ranking of allowable moves, selection will decide upon one single move to enact. In many algorithms selection is greedy, and simply takes the move with the best (highest or lowest) preference value. This is one example of a selection heuristic, another is to probabilistically select any one of the moves, in proportion to its preference value as compared with all other moves. It is also possible for an algorithm to make no selection, and not modify the solution vector during the current iteration. This behaviour can be observed in some perturbation methods (notably SAPS), where there is an expectation that the meta-level activities of the algorithm will modify the search space in such a way that the situation does not recur indefinitely.

One important point is that selection operates using only the observed preference values, and not by using any additional information about specific moves not already incorporated into the preference value. Another way of describing this is to say that selection is anonymous, having no access to information about which particular move a preference value refers to. From the point of view of selection it therefore makes no difference which one of a group of equivalently preferred moves is selected.

**Example:** A diagrammatic example of the operation of such a system will now be presented. Consider the problem of determining a 3-colouring of the graph below (Figure 2.1), using a complete search routine driven by a highest-forward degree ordering heuristic (number of uninstantiated neighbours). The variables are the 4 nodes, and the domain values are the available colours: red, green, blue. Consider that the algorithm will be a complete search procedure, so the initial assignment is empty.
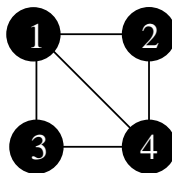


Figure 2.1: Example graph colouring problem

The most general contention heuristic (for a complete search routine) is to consider as candidates all moves involving currently unassigned variables.

Since candidates are to be selected according to highest forward degree, the choice for

the preference heuristic must elicit degree information about each move. In this case, the preference heuristic will simply assign to each candidate move the forward degree of that move's variable. As forward degree is independent of the value to be assigned by a move, all moves involving the same variable will be ranked equivalently.

Finally selection will choose a single move with maximum preference value as the move to enact. Since there will be moves with equivalent preference values, ties will be broken at random.

*Iteration 1:* As all variables begin uninstantiated, all moves are in contention in this iteration. The list of these moves and their corresponding forward-degree preference values are shown in Table 2.1. Of the moves with maximal preference value, select chooses one at random, say $1 \leftarrow red$. Meta-level processes now instantiate this choice; store the other potential values for variable $1$ should this instantiation fail; and recurse into the search procedure for iteration 2.



|  |  |  |  |
|---|---|---|---|
| $1 \leftarrow red = 3$ | $2 \leftarrow red = 2$ | $3 \leftarrow red = 2$ | $4 \leftarrow red = 3$ |
| $1 \leftarrow green = 3$ | $2 \leftarrow green = 2$ | $3 \leftarrow green = 2$ | $4 \leftarrow green = 3$ |
| $1 \leftarrow blue = 3$ | $2 \leftarrow blue = 2$ | $3 \leftarrow blue = 2$ | $4 \leftarrow blue = 3$ |

Table 2.1: Current state and moves in contention during Iteration 1 of graph colouring example.

*Iteration 2:* As variable $1$ is now instantiated, contention eliminates moves involving that variable from consideration. The moves remaining in contention along with their preference values are shown in Table 2.2, of which only those involving variable $4$ have maximal preference values. Select chooses say $4 \leftarrow red$, which immediately fails forcing reassignment to the next most preferred (but in this case equivalent) move, $4 \leftarrow green$.



|  |  |  |
|---|---|---|
| $2 \leftarrow red = 1$ | $3 \leftarrow red = 1$ | $4 \leftarrow red = 2$ |
| $2 \leftarrow green = 1$ | $3 \leftarrow green = 1$ | $4 \leftarrow green = 2$ |
| $2 \leftarrow blue = 1$ | $3 \leftarrow blue = 1$ | $4 \leftarrow blue = 2$ |

Table 2.2: Current state and moves in contention during Iteration 2 of graph colouring example.

*Iterations 3 and 4:* Only moves involving variables $2$ and $3$ remain in contention during Iteration 3 (shown in Table 2.3). All are preferred equally, and select chooses $2 \leftarrow blue$ at random, which is consistent with the previous instantiations. Iteration 4 then brings about the final assignment, of which the only consistent move is $3 \leftarrow blue$, which will eventually be discovered even if not initially selected.

$$2 \leftarrow red = 0 \qquad 3 \leftarrow red = 0$$
$$2 \leftarrow green = 0 \qquad 3 \leftarrow green = 0$$
$$2 \leftarrow blue = 0 \qquad 3 \leftarrow blue = 0$$

Table 2.3: Current state and moves in contention at beginning of Iteration 3 of graph colouring example.

### 2.6.1 Extensions

The following extensions are presented to demonstrate the generality of this representation, although they shall not be considered further in this current work.

**As used with an algorithm portfolio**

It can be seen that a representation for a single algorithm can be trivially extended to the case of a set of algorithms, each member of which operates independently on its own solution to an instance.

**As used with a genetic algorithm**

Each chromosome in a GA population may be considered a variable, even though chromosomes are themselves composed of assignments to a number of actual problem variables. Without loss of generality, consider the case of a genetic algorithm with a fixed-size population $\mathbb{P}$, as the case of a variable-size population can be seen simply as an arbitrarily large, but fixed-size population, where some elements are the 'null' chromosome.

The main difference that must be considered between a genetic algorithm and other types of search is that the allowable *moves* in a genetic algorithm are dynamic: they depend on the current assignments to the other variables (chromosomes) in the problem. Rather than being reassigned, a variable (chromosome) is replaced by the offspring of two parent chromosomes. Whereas the universal set for contention was previously the set of reassignments it is now the set of possible replacements, consisting of $2 \times |\mathbb{P}| \times |\mathbb{P}| \times L$ possible replacements, $L$ being the length of each chromosome, and each pairing producing two possible offspring.

Preference ranks each possible replacement by the fitness values of the respective parents, being the probability that those two chromosomes were selected to reproduce. Selection may operate either deterministically, selecting one of the $L$ random cut-points for the pairing with the highest probability of replacement, or probabilistically, randomly selecting from amongst all possible replacements based on the probability of each.

Mutation may be viewed similarly, by considering that each chromosome might be paired (using the exclusive-or operator) with one of a fixed set of mutation chromosomes, $\mathbb{M}$. The size of this set will be determined by the extent of allowable mutations, i.e. the maximum

number of simultaneous bit modifications, $n$. $\mathbb{M}$ contains all possible ways of selecting $n$ bits from $L$. Possible replacements from mutation may simply be included with those possible from recombination and assigned a commensurate preference value.

**As used with consistency methods**

As well as the constraint checks that are part of any backtracking procedure, regular consistency checks ensure that a search does not proceed down a path that cannot lead to a solution. Consistency checking methods can be viewed as meta-level activities of a search procedure. As an example of such a separation, consider the Davis-Putnam algorithm (Davis et al., 1962), which performs unit propagation and monotone variable fixing quite independently of its branching procedure. Unit propagation is certainly a form of forward checking (Génisson and Jégou, 1996), whilst monotone variable fixing is simply the end result of domain pruning.

Alternatively, consistency enforcing methods could be used as part of the contention heuristic to ensure that moves leading to a contradiction are never enacted.

# Chapter 3

# Adaptive Constraint Satisfaction

Many of the algorithms described in the preceding chapter included a parameter set with which their behaviour could be 'adapted' to better address specific classes of problems. But since none of those algorithms (with the exception of RSAPS) included an internal adaptation mechanism, any adaptation to a particular problem or class of problems is effected by an end-user and not by the system itself. This is not to say that the method of adaptation must be internalised within the constraint algorithm: many adaptive constraint systems employ methods that are quite separate from the actual constraint solvers they adapt. However, what all methods of adaptive constraint satisfaction have in common is that they seek to automatically adapt the behaviour of an algorithm to the problem or problem instance with which it is faced.

An existing method of classifying adaptive systems separates them into either *online* or *a priori* (offline) methods (Hutter and Hamadi, 2005). Such a categorisation makes an important distinction between what are often quite disparate and incomparable methods. Online methods attempt to recognise (during search) properties of an instance that might permit resolution of that instance. In some respects this is analogous to constraint weighting methods that attempt to learn which constraints are the most important to satisfy in order to escape a local minimum. The information learned is applicable only to that instance, and perhaps only to a particular part of the search space of the instance. In contrast, offline methods attempt to recognise (after the search) properties of an instance that are indicative of the broader class of problems to which the instance belongs, and which might be exploited when solving other problem instances of that class. To prevent the adaptive system from over-fittting to the training instances, recognising class properties requires information learned from a number of representative training instances. Because of this, learning properties of a class of problems is generally only realisable in offline methods, as there is no guarantee as to the amount or quality of information that might be available during the

course of trying to solve any one particular instance.

The categorisation of adaptive methods as either online or offline reveals the general intent of the adaptation as either to better solve a single instance or a broader class of problems. What such a categorisation lacks however, is any reference to the expressiveness of an adaptive system. That is, what types of adaptation is the system capable of?

In this work, an alternate method of categorisation is being introduced, which divides adaptive systems according to the extent of the adaptation they are able to effect upon an algorithm. Some adaptive systems embody only a fixed space of possible algorithms. Given such a fixed space, their method of adaptation amounts to selecting the most appropriate for the instance or problem class of interest from amongst this (fixed) set of pre-specified alternatives. These methods are not learning new algorithms nor adapting old ones, but instead performing a matching of algorithms to problems. As such methods are selecting from amongst a set of alternatives, they are termed *selective* methods. In contrast, other adaptive systems are not restricted to a fixed, or pre-specified set of algorithms, from which to select candidate algorithms. Whilst there is still a space of realisable algorithms (i.e. it still may not include all possible algorithms), the space is essentially unbounded: i.e. beyond satisfying the conditions for existing within that space, no *a priori* limit is placed on the size, form or complexity of the algorithms within it. Adaptive systems of this type are not so much selecting from out of this space as they are creating algorithms known to exist within it. For this reason, such methods are termed *creative*.

These two categorisations are by no means incompatible, but the second is favoured here as it better describes the fundamental capabilities of an adaptive system, rather than just the system's applicability to either individual instances or a larger class of problem. Furthermore, as the focus of this work is primarily offline methods, a division into online versus offline methods is unnecessary in the current context. In the following two sections existing methods of adaptive constraint satisfaction are characterised as either selective or creative. Following this is a discussion of a number of other characteristics that can be exhibited by an adaptive system.

## 3.1   Selective Approaches

This section details adaptive methods that seek to adapt by selecting from a portfolio of algorithms or by modifying a fixed-length parameter.

### 3.1.1 Chains of heuristics

Borrett et al. defined adaptive constraint satisfaction as an exclusively online process, "allowing for the active modification or switching of algorithms during the search process." (Borrett et al., 1996). It is not surprising then, that the adaptive system they proposed was an online method.

The stated motivation of their work was to try and prevent the thrashing behaviour that can detrimentally affect complete search procedures when faced with difficult problem instances. Their REBA (Reduced Exceptional Behaviour Algorithm) method involves a pre-specified chain of (complete search) heuristics and a "thrashing predictor" to recognise when the current heuristic is likely to begin (or has begun) to thrash. When such behaviour is recognised, the system applies the next algorithm in the chain of heuristics in the hope that it is not susceptible to the particular problem structure causing the identified behaviour in the current heuristic. The intention of such a method is that the number of occurrences of exceptional (heavy-tailed) behaviour may be reduced (hence the name given to this method).

In contrast to using a pre-specified chain of heuristics, it is possible to use a genetic algorithm to evolve a chain of heuristics appropriate to a particular problem class. This has been termed a Hyper-Heuristic Genetic Algorithm (HHGA) (Han and Kendall, 2003). A chromosome in such a system defines the order in which heuristics should be applied. The set of algorithms realisable with such a system is therefore the set of orderings that can be defined by a chromosome.

In HHGA, every algorithm defined within a current population is applied to a specific instance in an attempt to improve the quality of the solution to the current instance. The amount of improvement a chromosome effects on the current best solution to the optimisation problem is used as the measure of fitness for that chromosome for the purposes of reproduction.

As HHGA is concerned with specific problem instances it is an online approach, which is rather surprising given the computational cost of population-based methods. One possible explanation is, given the optimisation (rather than decision) nature of the test problems, the authors may have considered the speed of the algorithm to be less important than the quality of the solution located.

Although a system which adapts the ordering of its heuristic chain could be viewed as more powerful than one that relies upon a single fixed chain of heuristics, both systems have a bounded set of heuristics that they can apply. Like all selective approaches, such systems are constrained by the set of heuristics known to them, and there is nothing the system can

do should none of its heuristics prove efficacious on the particular problem instance with which it is faced.

## 3.1.2   Empirical Hardness Models

An *empirical hardness model* (Leyton-Brown et al., 2005) is a model of the expected amount of time that an algorithm will take to solve a problem instance. Machine learning techniques are used to construct a model of the hardness of an instance, in terms of its various features. The choice of machine learning technique is arbitrary, although Leyton-Brown et al. chose statistical regression. By constructing similar models for different algorithms, a system can subsequently identify the algorithm with the minimum expected run-time for the problem with which it is faced.

Whereas the work of Leyton-Brown et al. was concerned with complete search methods, models of the empirical hardness of problem instances are also appropriate for local search methods (Hutter and Hamadi, 2005). Specifically, this work demonstrated that the run-length of a local search procedure (in this case, the well-known SAPS algorithm (Hutter et al., 2002)) could be satisfactorily predicted from an analysis of various measures of the problem instance. Most of these measures were those identified in (Nudelman et al., 2004), but some additional features were introduced, based on what could be described as *probing runs* of a local search procedure: multiple independent short runs, intended to quickly gather information about the cost surface and the behaviour of an algorithm.

This work is further differentiated from its predecessor in that it learns models of the predicted run length of an instance given its problem measures, but also includes the parameter configuration of the search algorithm as additional model parameters. The advantage of such an approach is that the model can make predictions regarding previously unseen algorithm settings. This has important implications when an algorithm has a parameter(s) with a continuous domain, which would otherwise present an infinite number of potential algorithm configurations. To determine the appropriate parameter settings for a given instance, it is necessary to search the cost surface of the model for the algorithm parameters that minimise the expected run-time.

## 3.1.3   Other selective methods

As an adaptive system may begin with little fundamental knowledge about the heuristics that will be most effective on the problem it is required to solve, it is likely that it will have to consider heuristics that are not particularly effective on the target problem. In the case of an adaptive system that is simply selecting a heuristic from a set of alternatives, it

is desirable that such poor heuristics are recognised as soon as possible and removed from further consideration. This allows the computational requirements of the selection process to be reduced, or for the same computational resources to be more efficiently allocated to the better performing heuristics. However, this process is complicated by the fact that the run-time of non-deterministic algorithms can vary greatly, even on the same instance.

To balance these competing requirements a *racing algorithm* for configuring meta-heuristics was proposed (Birattari et al., 2002). The method works by iteratively evaluating all remaining candidate heuristics on a previously unseen problem instance. The performance results of each heuristic on the new instance are combined with its past results. A statistical test (specifically, the Friedman test) is then used to identify any candidates that are statistically sub-optimal, which are subsequently eliminated from further consideration. The procedure terminates once computational resources are exhausted, or one candidate has been statistically demonstrated to be the best.

The racing algorithm is a purely selective approach, and does not attempt to modify any of the heuristics it operates with. Furthermore, as it examines all candidate heuristics, it is only applicable when there is a finite set of heuristics.

Another selective approach was described by Nayerek (Nareyek, 2001), the premise of which is that a heuristic's past performance is indicative of its future performance within the scope of the same sub-problem. Each constraint is considered a sub-problem, and has a cost function and a set of associated heuristics. A utility value for each heuristic records its past success in improving its constraint's cost function, and provides an expectation of its future usefulness. Heuristics are in no way modified by the system, and their association to a problem's constraints must be determined *a priori* by the developer. A further difficulty with this approach is that, as the heuristic assigned to each constraint has (in each iteration) effective autonomy over the variables associated with that constraint. This means that in the general case, this method is unsuitable for use with complete search routines, as guarantees of completeness cannot be enforced.

### 3.1.4 Online Methods

There is no fundamental reason why an online adaptive method could not be creative (and learn a specific algorithm for a particular problem instance) rather than select an existing algorithm or combination thereof. There are a number of practical reasons why online methods are generally selective, which is why they are being introduced alongside selective approaches.

The first reason is that online approaches have only one opportunity to locate a solu-

tion. They are intended to locate solutions to individual instances, and once a solution has been found must necessarily terminate. Repeated trials (post-solution) would provide the algorithm with prior knowledge about the instance and potentially even its solution. The computational resources expended to derive that information cannot be ignored, so the performance of an online method cannot improve subsequent to locating a solution. Therefore, the motivation for any further adaptation to that instance is eliminated once a solution has been found. And although it is certainly possible to create new algorithms during that first trial, until a solution has been found an adaptive system has only uncertain, heuristic information about whether it is actually improving.

Another reason why online methods are generally selective is the difficulty in learning plausible, let alone effective, new heuristics for a problem. Many heuristics will not be suitable for a particular instance (or even problem class) and it can be difficult to ascertain that a heuristic is not working, once again, because the adaptive system can only make a heuristic assessment of the generated heuristic's performance. Many online methods therefore use a selection of handpicked heuristics that are already known to be effective on at least some problems of potential interest.

One final consideration is the cost-benefit ratio of the adaptation strategy. In an online method, anything learned applies only to a single instance, so the training cost can be recouped solely by improved performance on that instance. This necessitates that online approaches use much simpler adaptation strategies than those used in offline aproaches, when the cost of adaptation can be pro-rated over all of the instances the learned heuristic will eventually solve. The lack of online, creative methods in the literature should be considered as evidence that such methods are too costly to justify their use.

AdaptNovelty (Hoos, 2002) and RSAPS (Hutter et al., 2002) are two examples of online methods, promoted more for their ability to dispense with some of the parameters of their underlying heuristics. In the case of AdaptNovelty, it is the noise parameter that is adjusted automatically. Beginning with a setting of 0, which makes the algorithm maximally deterministic, but which can be adjusted upwards if the algorithm detects that it is has become trapped in a particular region (or local minimum). The metric used to assess stagnation is simply the average number of unsatisfied clauses in recent steps, which although not necessarily indicative of stagnation, is reported to give satisfactory results. The facility also exists to gradually decrease the noise parameter over time, once it has served its purpose in allowing the algorithm to escape from the troublesome region. RSAPS employs a similar method to automatically adapt its smoothing probability, although this parameter must be decreased rather than increased to cause the desired effect. Although in each case the

adaptive mechanism introduces more parameters than it removes, the introduced parameters are more stable than those that are replaced, resulting in a net (practical) decrease in the number of parameters of the algorithm.

## 3.2 Creative Approaches or Combinatoric Approaches

The section details adaptive methods that seek to adapt by learning 'new' algorithms.

For consistency between approaches, the following terminology will be used. An *algorithm* is an overarching control strategy that may be used to solve a constraint problem, (i.e. there are complete search algorithms and local search algorithms). Algorithms (those described here at least) are guided by *heuristics*, which indicate which variable-value assignment the algorithm should make at a decision point. Heuristics are themselves composed of simpler constructs, termed *measures*, which describe specific features of the current partial or infeasible solution to a problem instance.

### 3.2.1 MULTI-TAC

The MULTI-TAC system proposed by Minton (Minton, 1993, 1996) is designed to automatically synthesise domain-specific heuristics for solving CSPs, from more generic (but less efficient) heuristics. There are two stages of operation in MULTI-TAC: the first creates problem specific versions of generic rules appropriate for the problems in question; the second learns new heuristic combinations of these rules to better guide a search.

Two methods of specialising rules are described for use with MULTI-TAC. Domain-specific heuristics may be extrapolated analytically from the application of "meta-level theories" to generic rules that describe properties of partial solutions to CSPs. Alternatively, new rules can be synthesised through a brute force inductive approach by enumerating all possible rules of size 1, 2, etc. The two methods are described as complementary, as they lead to different types of specialisations. Whereas the analytic approach has a smaller branching factor and can therefore generate fairly complicated rules, it will occasionally overlook a much simpler rule easily located with the inductive approach, because such a specialisation did not logically follow from the supplied meta-level theories.

Four generic rules are explicated for use with MULTI-TAC's analytic learning engine (Minton, 1993): *Most Constrained Variable First*; *Most Constraining Variable First*; *Least Constraining Value First*, and; *Dependency Directed Backtracking*[1]. Rules are described using a form of first-order logic, and can include a variety of functions relevant to the domain

---

[1]Dependency Directed Backtracking was not included as a generic rule in (Minton, 1996), perhaps as it can be viewed less as a heuristic than as a part of the overall control strategy of the search algorithm.

of CSPs (e.g. *Satisfies, Adjacent, Assigned* etc.). The same language for rules is used by the inductive learning engine, but the combinatorial nature of this type of search places practical limits on how far such a search may be allowed to proceed. In MULTI-TAC, these rules lead to variable and value ordering heuristics specifically for complete (backtracking) search.

MULTI-TAC determines the most promising candidate rules by comparing their performance against that of the generic rules. A CSP solver (not guided by any of the newly generated rules) is run, but stopped occasionally at random. At these points, the generic rule is used to divide the space of possible variable-value assignments into best and worst choices, according to that rule. The new candidate rules are then evaluated to see if they make a similar division of the assignment space. As candidate rules are rarely perfect, all rules that perform correctly a majority of the time are retained.

The second stage of MULTI-TAC is to determine an efficient heuristic for the training instances, by learning applicable combinations of the newly specialised rules. Adaptation is accomplished by way of a beam search routine, which at every iteration increases the specificity of the current candidate heuristics. Any more-specific rule that does not perform as well as its less-specific progenitor is not considered further. Additionally, as part of the beam search, only the best $B$ expanded heuristics are retained for further extension during the next iteration. Search ceases when no rules outperform their progenitors.

A deficiency of MULTI-TAC is that the specialised heuristics are limited in size because their number grows exponentially with increasing size. Although this is controlled, in part, through the use of the beam search, this has its own disadvantages. The removal of under-performing heuristics from the population may eliminate those that exhibit poor individual performance but which would have performed synergistically if permitted further extension.

MULTI-TAC is appropriate for use on both satisfiable and over-constrained instances, and could also be used to specialise local, rather than backtracking, search heuristics.

### 3.2.2   The Adaptive Constraint Engine

Epstein et al. proposed the Adaptive Constraint Engine (ACE) (Epstein et al., 2002) as a system for learning search order heuristics. ACE doesn't directly learn new search order heuristics like MULTI-TAC, instead it determines problem-specific weights to associate with each of its "advisors" (measures). The weighted sum of the measures determines the evaluation order of variables and values. Such weights are indicative of the importance of that particular measure to the problems in question. Many different measures are considered, including: *Number of Values Remaining*; *Forward Degree*; *Resulting Domain Size of Neighbours* etc. The system can use the product of two measures as an additional measure,

providing it with its own associated weight, which is combined into the weighted sum like any other.

ACE is only applicable for use with complete search, as a trace of the expanded search tree is necessary to update the measure weights. Using this trace, measures that recommended paths that necessitated backtracking (i.e. that lead the search to inconsistencies), have their associated weight decreased in proportion to the length of the path required to realise the incorrect assignment. Similarly, the weights on measures that lead the search towards a solution are reinforced.

ACE is described as adjusting the weight associated with each measure "after a problem has been sovled successfully", which prohibits it from learning in situations where it is unable to locate a solution. The alternative would be to update measure weights at the time the backtrack occurs, permitting later stages of the search to benefit from the knowledge learned so far. Penalising measures would still be possible under such a scheme, although it would be difficult to raise measure weights until a correct path to solution was known. One alternative would be to redistribute weight from the offending measure onto the others, rather than simply subtracting it.

A novel feature of the system is that it can learn different heuristics for different stages of the search. Three different stages of search are identified: early (fewer than 20% of variables instantiated), middle (between 20% and 80% of variables instantiated) and late (more than 80% of variables instantiated).

In practice ACE has only a limited ability to learn new heuristics: although it can multiplicatively combine exactly two measures, and can learn appropriate weights for combined measures, only 4 of its 19 measures were allowed to participate in such combinations. There is no fundamental reason why ACE must be restricted in this way, as the use of feedback remains appropriate even when the number of combinations is increased. Hence, the most likely reason for this restriction is the practical consideration of the resulting combinatorial explosion of potential heuristics.

Although ACE is described as being applicable to over-constrained problems, it does not obviously follow how its weight update scheme would apply when every search path eventually derives an inconsistency. ACE exhibits one significant advantage over MULTI-TAC, in that its use of feedback to update weights facilitates the identification of synergies between measures.

### 3.2.3   Composer

Composer (Gratch and DeJong, 1992, 1996) is an adaptive system for specialising planning and scheduling algorithms. In Composer, "control strategies" (heuristics) are incrementally extended one "rule" (measure) at a time, using a hill-climbing approach. In much the same way as ACE, Composer analyses a trace of the path to solution taken by the algorithm, to extract a utility value for each measure, by identifying the search paths that could have been avoided by the use of that measure. Such a method is similarly unable to learn from situations where no solution is found by the original algorithm. Particular implementations of the Composer system, specifically in (Gratch and Chien, 1996), have overcome this limitation by applying the system to partial solutions of an overall problem.

Composer repeatedly solves problem instances from the training set, obtaining statistics about the utility of each candidate measure. Once Composer's confidence in its measure exceeds a threshold, the measure with the highest positive utility (if any) is adopted as part of the search heuristic. As all statistics collected are dependent on the current heuristic, the adoption of an additional measure into the heuristic necessitates that all previous statistics about measure utility are discarded. Measures that Composer is confident exhibit negative utility are removed from the candidate set, but since they are not part of the heuristic their removal does not require the discarding of current statistics. This type of adaptation can be seen as a form of beam search, with beam width 1, and is unable to identify synergies between heuristics.

### 3.2.4   ADATE

ADATE (Olsson, 1995) is a system for automated functional programming, which generates new programs by transforming existing ones. ADATE uses fitness-like measures of program quality such as correctness, syntantic complexity and time complexity in order to guide the search through the space of possible programs. However, Olsson distinguishes ADATE from evolutionary methods on the grounds that "the latter are very poor at inferring recursive programs since they use primitive program transformations and an unsystematic search of the program space" (Olsson, 1995).

Four basic types of transformation are proposed for use with ADATE: replacement, abstraction, case-distribution, and embedding. The respective roles of these transformations are as follows. Replacement overwrites one sub-expression of a program with a newly synthesised expression. Elements of the sub-expression being replaced may be re-incorporated into the synthesised expression. Abstraction defines a new function to be an old function

where some elements have been replaced by arguments of the new function. Case-distribution transforms a function accepting a series of cases as arguments into a function where the cases are part of the function body itself. Embedding changes the type definition of a function, such as by incorporating an additional argument into the function or by changing the type of one of the arguments of the function. A number of rules for "compound transformations" (Olsson, 1995) are also described, by which these individual transformations may be made more complex.

Apart from the fact that ADATE uses "transformations" rather than genetic operators it is otherwise very similar to an evolutionary algorithm. It maintains a population of programs, although this population initially contains just a single candidate - the undefined element - which is to be expanded. Candidate programs are ranked according to both a "program evaluation function" (analogous to fitness) and the number of case expressions they contain. The method by which ADATE transforms programs is more deterministic than an evolutionary algorithm however, with fitter programs guaranteed to be expanded first.

A potential problem with ADATE is that, like MULTI-TAC, there is an expectation that child programs are at least as good as their ancestors, and any that are not are immediately discarded. Whilst such behaviour might be justified by the view that evolution takes no backward steps, it neglects one very important concept in computational optimisation: local optima. There is no reason to believe that a program that does not out-perform its ancestors is necessarily inferior, and could not be improved with future transformations.

### 3.2.5 CLASS

The CLASS system (Fukunaga, 2002, 2004) (an acronym for *Composite heuristic Learning Algorithm for SAT Search*) is an evolutionary-style approach for learning new heuristics for local search satisfiability testing algorithms. CLASS constructs heuristics composed mainly of *if..then* style production rules and relies heavily on measures that have been used in the GSAT, WALKSAT and NOVELTY families.

The system randomly generates an initial population of heuristics, evaluating the perfomance of each to determine which will be selected to participate in reproduction. New heuristics are developed exclusively using the *composition* operator, which is a special-purpose operator not dissimilar to those of ADATE (Olsson, 1995). This operator works by taking two existing heuristics $H_1$ and $H_2$ and a boolean condition $C$ to create new heuristics of the general form "If $C$ then $H_1$ else $H_2$". Ten different conditions were explicated for use with CLASS, as shown in Table 3.1.

1-5. (If (Rand p) H1 H2) for p=10, 25, 50, 75, 90 - probabilistically chooses between heuristics.

6. OlderVar H1 H2 - chooses the heuristic that prefers the oldest variable.

7. (IfTabu 5 H1 H2) - will use heuristic 2 if heuristic 1's choice is tabu.

8. (IfNegGain0 H1 H2) - uses heuristic 1 only if it won't dissatisfy any constraints.

9. (IfVarCompareNegGain ($\leq$) H1 H2) - prefers the choice that dissatisfies fewest constraints.

10. (IfVarCompareNetGain ($\leq$) H1 H2) - prefers the choice offering the most improvement.

Table 3.1: The 10 different compositions of CLASS

There is an important distinction between the composition operator of CLASS and the operators traditionally used in other evolutionary algorithms, like the genetic algorithm (Holland, 1975) and genetic programming (Koza, 1992), and that is whether the operator disrupts a parent individual's genetic material during reproduction. In the standard genetic algorithm, a fixed-length chromosome necessitates some disruption to a parent individual's chromosome if evolution is to occur. However, when a chromosome is an expression without a predetermined size or structure (as in genetic programming and CLASS), disruption is not necessary and it becomes possible to incorporate all of a parent's genetic material into its children.

The composition operator used in CLASS does not (inherently) disrupt a parent's genetic material when forming a child. The two parents are included in the new child in their entirety (along with a boolean condition as outlined previously). This is in contrast to the behaviour of the standard operators in genetic programming, which do disrupt the parents' chromosomes. Whilst it is advantageous to prevent disruption to, and hence the removal of, above-average substructures from the population, there are a number of associated disadvantages with such a method. Of prime importance is the exponential growth in the size of each new child resulting from combining two entire parents. Secondly, as no genetic material is ever removed from a candidate expression, redundant genetic information that might once have exhibited coincidental above-average performance will remain present for all time[2].

Rank-based selection (Baker, 1987) is used to select the parent heuristics to participate in composition, as well as to determine which heuristics in the population are to be replaced by the newly generated child heuristics. The identification of synergies is not precluded, as selection is probabilistic in both cases. The reasons for the use of rank-based selection in CLASS was not explicitly stated. One possible motivation for its use is the large performance variance often observed between the better and worse elements of a heuristic population, which can unfairly prejudice selection away from the worse elements. The use of rank-based

---

[2]These limitations were addressed in part in (Fukunaga, 2004) by replacing subtrees that would exceed the size bound with randomly generated leaf nodes of a corresponding type.

selection could therefore make the selection process more egalitarian, but at the same time can exaggerate differences between comparatively similar performance (Koza, 1992).

### 3.2.6 SALSA Meta-Heuristic Factory

SALSA (Laburthe and Caseau, 1998) is a language for specifying search algorithms. It has the capability to represent local, global and even hybrid search algorithms. One of the main advantages of SALSA is that it dramatically simplifies the specification of complex optimisation procedures, abstracting away many of the underlying Constraint Programming activities.

The operation of SALSA is to create a separate *process* for each node visited (a node can be a node in the usual sense as part of a search tree or alternatively refer to a state visited in an incomplete procedure). The power of SALSA to represent such a variety of search routines is a consequence of the lack of synchronisation between these processes. However, they are not fully independent as all processes have access to global information about the state of the overall search. Execution occurs by way of a formal set of transition rules, written in a calculus of distributed processes.

Because of the variety and ease of representing disparate algorithms in SALSA, it was used as the basis of a representation for an adaptive method to learn new algorithms for the vehicle routing problem with time windows (Caseau et al., 1999). Since the expressiveness offered by SALSA is so high, SALSA was reduced to a specialised algebra exclusively for representing hybrid routing algorithms. Primitives in the algebra denote various search routines and post-optimisation steps that may be used as part of more complex strategies.

Candidate expressions are initially generated randomly, with each primitive selected according to a user-defined probability distribution. Subsequent modifications are made by a process of *mutation*, which can make a modifications of varying severity to a candidate expression. A *shallow modification* is only permitted to make small changes to the numeric constants in an expression. An *average modification* allows larger changes to numeric constants and some interchange in functional terms. Finally, a *deep modification* allows substitution of one term by a completely different term, subject to a given probability distribution.

The learning process is iterative. After the initial randomly generated population, each subsequent population is composed of the best performing expressions form the previous iteration as well as mutated versions of these expressions. Adaptation ceases when computational resources are exhausted.

## 3.3    Five Desirable Features of Adaptive Systems

Having examined the existing methods of adaptive constraint satisfaction, it is possible now to consider the strengths and weaknesses of the various approaches to identify the desirable characteristics of an adaptive system. The five characteristics discussed here were originally proposed in related work (Bain et al., 2004b), albeit using slightly different terminology. The case for the importance of each feature to the effectiveness of an adaptive constraint system[3] is argued individually for each, with citations to independent sources used wherever possible.

### 3.3.1    Versatility

The *versatility* of an adaptive constraint system refers to its ability to adapt both local and complete search heuristics, but also its suitability for adapting to both satisfiable and over-constrained problems.

It can certainly be argued that there is no reason for an adaptive system to be versatile. One alternative is to have two such systems, one able to adapt local search routines and another, complete search (and similarly for satisfiable and over-constrained instances). Whilst it is possible that such an arrangement could offer performance benefits, the existence of two disparate systems means that advances in one method of adaptation would not necessarily translate to the other.

Within the domain of constraint satisfaction, specifically for propositional satisfiability testing, the utility of using both complete and local search procedures is most recently evidenced by the results of the SAT 2005 competition (Le Berre and Simon, 2005b). In this competition, problem instances from three categories were considered: random, crafted and industrial. A local search solver was demonstrated most effective on the random instances, whereas a complete solver was most successful on the crafted and industrial categories.

Unlike some of the other features, the versatility of a system (for both complete and local search procedures) is determined by both the chosen representation and the method of adaptation. Versatility with regard to satisfiable versus over-constrained problems depends solely on the method of adaptation, however.

Overall, versatility was observed in most of the adaptive systems considered. Although some systems were described as suitable for one domain, it is relatively simple to consider how their representation could be modified to suit the alternate type of algorithm. The

---

[3]Although the discussion of the crucial features relates specifically to adaptive constraint satisfaction, we posit that these arguments could easily generalise to other types of adaptive systems, with the possible exception of those for versatility.

notable exception to this is the representation used by Nayerek, where the association of heuristics with individual constraints which seek to satisfy themselves, presupposes a local search strategy.

In general, any method of adaptation that relies upon an external, overall assessment of an algorithm's performance will be versatile, as it is reasonably straightforward to substitute the performance measures appropriate for one domain with those for the other. This is in contrast to methods that seek to adapt by modifying an algorithm in response to its internal behaviour. This was observed with ACE, where adaptation attempts to correct poor branching decisions as an internal feature of the algorithm. As local search routines do not exhibit such features, there is no natural extension of such a method to local search, just as the method is not directly applicable to over-constrained problems where all search paths necessarily result in a contradiction.

### 3.3.2 Creativity

Although AI practitioners may disagree on what objectively constitutes new, novel or intelligent behaviour by an artificial system, one definition for artificial creativity is that it is "novel combinations of old ideas" (Boden, 1994). In the context of an adaptive constraint system then, creativity lies in the ability of the system to learn 'new' heuristics composed from lower-level ('old') measures. This concept, combined with expressivity, has been previously referred to as "plasticity" in other work (Minton, 1996).

Consider that a constraint algorithm can be described as a Turing Machine: the underlying heuristic of the algorithm is analagous to the machine's state transition rules table; any parameters of the procedure and a constraint instance are its data. The fundamental difference between selective and creative approaches is that selective approaches modify only the data on which a (pre-specified) set of rules operates, whereas creative approaches modify the rules themselves.

Creative methods are therefore distinguished from those that seek to adapt to different problems by either: (1) choosing from a set of completely specified heuristics/algorithms; (2) by modifying the order in which such heuristics/algorithms are applied, or; (3) by modifying a fixed length parameter of an algorithm.

**1.** The justification for the first two of these needs little explanation, as utilising a set of completely specified algorithms (without modification) is not creating anything - novel or otherwise. Although a system might identify that one algorithm is better than another on a problem in general (or on specific instances of that problem), that is no more creative than identifying that $1 < 2$.

**2.** In the second case, regarding learning an ordering of heuristics, such an ordering is not so much a combination as simply a sequence of entirely independent stages. However, if this argument is not sufficiently compelling, an ordering of heuristics can be viewed as a fixed-length parameter of an algorithm, in which case the arguments for case three apply.

**3.** Two arguments are presented against considering the tuning of a fixed-length parameter to be creative. The first (3a) is similar to Turing's imitation test of intelligence (Turing, 1950) whereas the second (3b) argues that although tuning procedures may appear to modify an algorithm, in actual fact they do not.

**3a.** The first argument is one of anthropomorphism, and is for creativity what the Turing test is for intelligence. An artificial system is 'creative' if it could at least appear to be creative in a human sense. So although parameter tuning is a computationally difficult problem, it would not be considered a creative endeavour if performed by a human. Therefore, it is similarly not creative if performed by a machine.

**3b.** The second argument relates to the potential expressitivity of algorithms where only a fixed-length parameter is learned. If a (numerical) parameter of the algorithm is of a fixed length, then an algorithm (with unbound parameters) may be considered to embody a finite space of completely specified algorithms (without unbound parameters). The tuning of fixed-length parameters is therefore simply the selection of one completely specified algorithm out of many, which has already been determined in argument 1) above not to constitute creativity.

Returning to the definition of a constraint algorithm as a turing machine, it could be argued that the specification of an algorithm could be encoded numerically, thereby reducing the algorithm to a single value. A universal constraint algorithm, analagous to a Universal Turing Machine, could then accept the encoded specification of the algorithm as a parameter and then behave as that algorithm. Viewed in this way, modifying a set of rules is functionally equivalent to tuning a single numeric parameter. Such an argument reduces the previously defined creativity in modifying a set of rules to mere selection. However, what this argument neglects is that there is a fundamental semantic distinction between the data passed to a procedure and the state transition rules of a procedure. The respective modifications of either of these are therefore not functionally equivalent - regardless of the encoding used for either. ◇

In conclusion, creative approaches are those that modify the underlying heuristic rules of a constraint system by generating novel combinations of lower-level measures. In contrast, selective approaches do not modify the underlying rules of the system. They modify only

the data passed to such a procedure, which has been shown to be equivalent to selecting one completely specified heuristic out of many.

### 3.3.3 Expressivity

Adaptive systems are declared to offer *expressivity* if they do not impose a pre-specified bound on the size of heuristics that may be learned. The absence of an *a priori* size bound allows the adaptive system to determine the size of algorithm necessary to appropriately adapt to the target problem. The expressivity offered by an adaptive system can be limited by either the representation or the method of adapation.

Bounds due to the representation can take a number of forms. The adaptive system may be simply tuning the fixed set of parameters of an underlying algorithm such as in AdaptNovelty. Fixed length chromosomal representations as used by genetic algorithms are another example of this form of representation. Although the information encoded by a chromosome may often be a 'black-box' to the method of adaptation, this need not be the case. An adaptive system may alternatively have a known (fixed) set of component measures that it is able to combine into a heuristic by way of a weighted sum, as was seen in ACE (Epstein et al., 2002), and where the influence of individual measures can be modified. These are all examples of representations constrained by an *a priori* expressivity bound.

Examples of systems where the representation is not a limiting factor in the expressivity, are MULTI-TAC (Minton, 1993), CLASS (Fukunaga, 2002) and the SALSA meta-heuristic factory (Laburthe and Caseau, 1998). Both of these systems use a functional expression to represent a heuristic, the size of which is not restricted and which may take any form consistent with the underlying measures that compose it.

The other reasons why an adaptive system may offer only limited expressivity relate to the method of adaptation rather than the underlying representation. Even if the system provides a suitable method of adaptation for its representation, other factors may limit the level of expressivity offered.

The adaptive method may create an exponentially increasing number of candidates, limiting the number of iterations the method may operate for. The effect of this is that candidate heuristic size is restricted to what can be realised within that number of iterations. MULTI-TAC is one example of this type of behaviour, although it is mitigated to some extent by culling underperforming candidates.

Alternatively, exponential growth may also be seen in the size of candidates, rather than their number. Methods that lead to such growth are similarly only effective for a limited number of generations, and further adaptation becomes impractical. The primary example

of this is CLASS, which combines two candidates in their entirety, leading to exponential growth in heuristic size. This was controlled in part by replacing large candidates with smaller, randomly generated ones.

In conclusion, an adaptive system can only offer unbounded expressivity if it provides both an unbounded representation and a method of adaptation appropriate to it. However, even if these requirements are met, the expressivity offered by an adaptive system may still be restricted. In practice, if the method of adaptation results in an exponential growth in the the size or number of candidate heuristics, then the expressivity will be constrained by the steps taken to artificially limit such growth.

### 3.3.4   Foresight

*Foresight* describes the property of an adaptive system whereby synergies between measures are recognised. Although measures may exhibit poor independent performance, they may perform well in conjunction with other measures. The difficulty arises from the exponential increase in potential candidates when measures are considered in combination. To control this growth, many adaptive methods automatically/unreservedly remove poor performing candidates from the population, discounting the possibility that they would have performed well when combined with another measure. Such systems are thus not exhibiting foresight. The corollary of this is that methods that perform well individually may not continue to do so when combined in tandem with another method.

Foresight is absent from methods that use purely elitist selection or beam search where only the best candidates are considered further. MULTI-TAC, Composer and the SALSA meta-heuristic factory all exhibit this limitation. Methods that employ feedback (such as ACE) or that make probabilistic selections such as CLASS, are able to identify synergies. This is because poor performing elements are not automatically excluded from further consideration.

The argument for foresight (apart from preserving potentially good candidates) is that it is necessary if truly automated programming systems are to be developed. That is, systems that without the involvement of any human expertise are able to develop an efficient solution for a particular problem. As measures are broken down into simpler and simpler constructs they are less and less likely to demonstrate good independent performance. The ability to see past such performance is therefore crucial to the performance of an adaptive system.

### 3.3.5 Hindsight

The final property to be discussed is termed *hindsight*, and describes the ability of an adaptive system to look back and learn from its own failure. Particularly as adaptive methods are likely to be applied to problems for which there are not already efficient algorithmic solutions, there is no reason to suppose that the behaviour of an adaptive system will, at least initially, locate a solution to the problem.

For this reason it is important that an adaptive system can recognise and exploit those methods that have resulted in at least some measurable progress towards a solution. Whilst this is likely to be only an estimate of the method's efficacy, it allows the adaptive system to continue the process of adaptation without having determined a solution. Also, if the estimation is correlated with actual performance then the learning time may be reduced, since the evaluation of a candidate method is not contingent upon waiting for the method to locate a solution.

The Composer and MULTI-TAC systems are capable of learning from failure, since each can gather statistics of various kinds to determine which measures appear to have merit. As described, CLASS would not learn from failure as the fitness of a heuristic depends only on successful runs, but the fitness measure could be modified to include features that discriminate between heuristics even when they fail to find a solution. ACE is also unable to learn from failure, as it does not update the measure weight profiles until it has completed exploration of the search tree, but a possible revision was already described in Section 3.2.2.

## 3.4 The Five Features and the Proposed Representation

Having identified five important features of an adaptive system, it is now possible to consider the algorithm representation proposed in the previous chapter in terms of them. As two of the features - foresight and hindsight - depend exclusively on the method of adaptation, they will not be considered here. How the remaining three features - versatility, creativity and expressitivity - are exhibited by the proposed representation will now be discussed.

Recall that in the proposed representation an algorithm is defined by a commitment to a heuristic for each of the three stages (contention, preference and selection) that occur during every iteration of a search procedure. All algorithms have necessary meta-level activities as well, but these are independent of the representation. The heuristics for each of the three stages are defined as functional expressions of the appropriate type for their particular stage. These heuristic expressions are composed of measures that describe the problem and the state of the search; mathematical and logical operators; and also constants.

This representation is versatile, as it allows the specification of both complete and local search heuristics. Although the types of heuristics that would prove effective in each case would differ, as would the set of appropriate measures, both complete and local search heuristics can be represented as functional expressions.

This representation is also creative, in that it does not just provide for the selection of one algorithm out of many, or for the tuning of a numeric parameter set. Instead it allows novel heuristics to be constructed out of combinations of simpler lower-level measures.

Lastly, this representation also offers expressitivity, as the expressions are not constrained to be of a particular size or form. Subject to the availability of a sufficiently diverse set of operators with which to combine the measures and constants, expressions may include any number of occurrences of each lower-level measure, in any (syntactically correct) arrangement imaginable.

The following chapter will introduce a method for modifying heuristics posed in this representation, after which the remaining features that depend on the method of adaptation will be discussed.

# Chapter 4

# Evolving Algorithms for Constraint Satisfaction

In the preceding chapter, five features that an adaptive constraint system may exhibit were discussed. The proposed algorithm representation was shown to exhibit the subset of features dependant on the representation of an adaptive constraint system. This chapter details genetic programming (by first introducing its predecessor, the genetic algorithm) and then demonstrates that it is a suitable method of adaptation for constraint algorithms. Genetic programming will then be shown to exhibit the remaining desirable features of an adaptive constraint system.

## 4.1 Evolutionary Algorithms

### 4.1.1 The Genetic Algorithm

The *genetic algorithm* (Holland, 1975) or *GA* is a local search method inspired by the processes of biological evolution. Unlike most of the local search algorithms discussed earlier, the genetic algorithm is a population-based method, maintaining at any given time a number of candidate solutions rather than a single solution. Methods analogous to those of natural selection and sexual reproduction allow the algorithm to explore solutions in the local area of existing solutions. What constitutes the local area depends on the available genetic operators and the composition of the population at the time.

Every element in the population of candidate solutions is termed a *chromosome*. Classically, chromosomes in a genetic algorithm are a fixed-length representation of a solution, and although variable-length (modifiable) representations were similarly described in (Holland, 1975), only the classical representation is described here. Each chromosome is composed of a sequence of *genes*, each specifying one variable or element of the solution. Each gene is likewise composed of as many *alleles* as required to precisely ascribe a value to that gene's

$generation \leftarrow 0$
Generate a population $\mathbb{P}$ of $n_p$ random solutions
**while** $generation < lastGeneration$ **do**
    Evaluate fitness of all solutions in $\mathbb{P}$
    $\mathbb{P}' \leftarrow getbest(\mathbb{P}, n_c)$
    $\mathbb{S} \leftarrow$ Select using fitness $n_b$ solutions from $\mathbb{P}$
    **while** $\mathbb{S} \neq \emptyset$ **do**
        Pick two random elements $x$ and $y$ from $\mathbb{S}$
        $\mathbb{S} \leftarrow \mathbb{S} - \{x, y\}$
        $\{x', y'\} \leftarrow crossover(x, y)$
        $\mathbb{P}' \leftarrow \mathbb{P}' \cup \{x', y'\}$
    **endwhile**
    $\mathbb{M} \leftarrow select(\mathbb{P}, n_m)$
    $\forall m \in \mathbb{M}, \mathbb{P}' \leftarrow \mathbb{P}' \cup \{mutate(m)\}$
    $\mathbb{P} \leftarrow \mathbb{P}'$
    $generation \leftarrow generation + 1$
**endwhile**

Figure 4.1: Evolutionary procedure using elitist cloning. $n_c, n_b$ and $n_m$ are the number of elements to clone, crossover and mutate respectively.

variable. Commonly each allele is a binary digit, each gene a binary number and each chromosome a sequence of binary numbers. This is by no means enforced or required, but is simply an artifact of working with digital computers.

A genetic algorithm is generally initialised with a population of random chromosomes. Beginning with this initial generation, the evolutionary process then iteratively repeats the following steps: evaluate the fitness of each element in the current population; repeatedly apply the genetic operators to produce new population elements; form a new population from these elements. An algorithm of the full procedure is given in Figure 4.1.

The evaluation of each chromosome is performed by a function mapping the chromosome (a value in genotype space) to some other value (in phenotype space). This mapping function may be dynamic, and the phenotype space may well be multi-dimensional. Despite this, the observed performance must eventually be reduced to a numeric measure of *fitness*, which will be used to calculate the probability that a particular chromosome will be selected to participate in one of the genetic operations that will constitute the subsequent generation. Selection of an appropriate fitness measure is crucial to the success of a genetic algorithm. An important point to note is that individual chromosomes of comparatively low fitness are not automatically excluded from operator selection, they are simply characterised by a lower probability of selection.

Traditionally evolutionary algorithms have been described as either $(\mu+\lambda)$ or $(\mu, \lambda)$ (Bäck et al., 1991). In the former, the $\mu$ parents produce $\lambda$ children and both $\mu + \lambda$ individuals participate in selection, permitting parents to live for further generations, subject to fitness-based competition with the offspring. Alternatively, the life span of an individual can be

restricted to exactly one generation, as it is in the $(\mu, \lambda)$ variant. After generating $\lambda$ offspring, the $\mu$ parents are automatically removed from the population, leaving only their offspring to compete for a position in the subsequent generation.

The primary genetic operator is termed *crossover*, and performs a recombination of genes from two individuals (the parents) to form two new individuals (the offspring) to become part of the next generation. To create the offspring, the operator selects a random position in the chromosomes of the two parents and swaps the genetic material after this position between the two chromosomes. The first offspring is composed of the earlier genes and alleles of one parent and the later genes and alleles of the other. Similarly, the second offspring is composed *vice versa*. A well-known variation of this operator is *multi-point crossover*, which generally selects two positions in the parent chromosomes instead and interchanges the alleles between these points. Such behaviour is akin to treating the chromosome as a ring, rather than a string, but despite any apparent benefits the no free lunch theorems dictate that it is no more effective on average than single-point crossover. However, it will often be the case that problem specific differences and other variations in the underlying algorithm make one crossover method more effective on particular types of problem.

Another important operator is that of *cloning*, which performs the exact duplication of a chromosome from the current generation. In many cases, selections made for this operator are *elitist* in that individuals with the highest observed fitness are selected automatically (rather than probabilistically) in preference to the less fit. Whilst elitism does serve to guarantee that fitter individuals are preserved in future generations, its overuse can reduce the genetic diversity of the population, potentially leading to premature convergence to a non-optimal solution. Elitism is not restricted to cloning but can be used when making selections for any operator.

The final operator that is traditionally used is that of *mutation*. Mutation serves to improve the diversity of a population by introducing particular alleles that were perhaps not present in the original population, or have since disappeared. Considering that the other operators do nothing more than rearrange existing alleles into new genes (and hence, chromosomes), mutation is the only operator of the three discussed that can introduce new material into a population. A chromosome selected for mutation will have one or more of its alleles changed, modifying it for future generations.

Subsequent generations of chromosomes continue to be evolved whilst time and computational resources permit. The most computationally expensive part of the procedure is the calculation of the fitness values of individual chromosomes. Since the fitness of each chromosome is independent of the population as a whole though, these calculations can occur in

parallel without any intermediary communication.

Having discussed how a genetic algorithm works, some mention of the reasons why it works is also necessary. Whilst it might be supposed from the foregoing discussion that the success of the evolutionary process stems from the explicit proliferation of successful chromosomes, this is only part of its operation. Implicit in the proliferation of successful chromosomes is the proliferation of successful *schemata*. Schemata are a generalisation of a chromosome, in which some of the alleles may have been replaced by a 'don't-care' symbol. Schemata are said to match any chromosomes where there is a correspondence between the alleles of the chromosome and those of the schemata, excepting any alleles occuring in the don't-care positions of the schemata.

Any particular chromosome will match a number of schemata exponential in the length of the chromosome. Similarly, each schemata will be matched by a number of different chromosomes in the population. This property of the system is termed *implicit parallelism*. Overall, each schemata will exhibit an average fitness due to the individual fitness contributions of the matched chromosomes. The predilection of the system to select above-average chromosomes is in reality a preference for above-average schemata, and over time the repeated selection of such chromosomes results in a greater number of copies of above-average schemata within the population. The true role of the genetic operators then, is to facilitate the testing of these schemata in different chromosomal contexts, to further ascertain any correlation they may have with above average fitness. This is accomplished without ever making an explicit calculation or evaluation of the performance of individual schemata.

This proliferation of above-average schemas through selective pressure will be mitigated by the disruption that the genetic operators cause to the selected chromosomes. As chromosomes are of a fixed length, their adaptation necessitates at least some disruption to the existing genetic material, and hence, a variation in the set of schemata matched by the modified chromosome. Since the underlying premise of the schema theory is that it is the proliferation of the underlying schemata that is important, the possibility of disrupting an element from matching a schemata must also be considered.

$$E\left[m\left(H, t+1\right)\right] \geq Mp\left(H, t\right) \times \left(1 - p_m\right)^{O(H)} \times \left[1 - p_{xo}\frac{L(H)}{N}\left(1 - p\left(H, t\right)\right)\right]$$

This equation (Poli, 2001a) represents a lower bound on the number of future occurrences of a schemata, since although disruption removes some schemata from the set of those matched by a chromosome, it must necessarily result in the modified chromosome matching some other schemata. $E\left[m\left(H, t+1\right)\right]$ is the expected number of chromosomes in

the population matching schema $H$ at time $t + 1$. The three terms on the the right-hand side respectively account for: 1) Loss due to non-selection, being the population size $M$ multiplied by $p(H, t)$, the probability of selection of schema $H$ at time $t$. 2) Disruption due to mutation, where $p_m$ is the probability of mutation and $O(H)$ is the number of non-*don't care* symbols in the schema. 3) Disruption due to crossover, where $p_{xo}$ is the probability of crossover, $L(H)$ is the defining length of the schema[1], and $N$ is the total length of the schema.

Some elements of this equation are difficult to quantify, such as the various $p$ terms, which depend on the average fitness of a particular schemata. Qualitatively though, it can be seen that provided a schemata offers an appreciable fitness benefit and has a relatively small defining length $L(H)$ then that particular schemata will proliferate within the population.

This conclusion attributes the performance of a genetic algorithm to the existence of *building blocks*, i.e. small clusters of genetic material that imbue an individual with above average performance in the phenotype space (Holland, 1992). An important consequence of this is that as the structure of the chromosome is (generally) determined by the user, the performance of the genetic algorithm is dependent upon the user's modelling of the problem. Although a schemata may exhibit above-average performance, an inappropriate modelling where its alleles are widely seperated on the chromosome could increase the probability of disruption to such an extent that the schemata does not proliferate within the population, despite its above-average fitness.

## 4.1.2  Genetic Programming

The genetic algorithm as described is appropriate when the general form of the solution is known in advance, as well as the necessary ranges of all variables. Without this information though, it is difficult to apply a genetic algorithm to the problem, necessitating an alternate representation of the problem.

That there was a need for a richer representation was recognised early, with Holland proposing alongside his original treatment of the method a *broadcast language* (Holland, 1975) that could be used to represent meta-level information about a domain. This extended the genetic algorithm to allow arbitrarily long chromosomes. Subsequently, the *messy genetic algorithm* was proposed (Goldberg et al., 1989) which also used variable-length chromosomes. Messy GAs combine relatively short, less complex substrings to evolve longer chromosomes that handle more complex aspects of a problem. Whilst both of these methods still worked

---

[1]The defining length of a GA schema is the distance between the outermost specified, non-*don't-care* alleles.

with linear, albeit variable-length chromosomes, it was Koza's *genetic programming* (Koza, 1990) that proposed the direct manipulation of a tree-like representation for a problem's solution.

Genetic programming (Koza, 1990, 1992, 1994) (GP) is an alternate method to GAs that does not require a specification of the form of a solution to a problem, nor information about appropriate ranges for a problem's variables. This is because where a traditional GA uses a static, linear data structure to represent a solution, the tree data structure used in GP is dynamic, with solutions continuing to increase in size whilst there is a fitness benefit to be gained. It is this that makes genetic programming more expressive than a fixed-length GA.

Genetic programming gives a search routine freedom to explore structurally different solutions to a problem. Whilst genetic programming can be susceptible to bloat - an increase in chromosome size or complexity with limited fitness benefit - allowing such structurally diverse solutions to a problem means that fewer assumptions are made by the user that could result in interesting solutions being overlooked (as alluded to in the preface).

Genetic programming does not absolve a practitioner of all responsibility when solving a problem. It is still necessary to specify sets of functions and terminals (constants) sufficient to solve the problem. It is these functions and terminals that will compose the chromosomes representing solutions to the target problem. Internal nodes will consist of parameterised functions, with terminals and 0-arity functions filling the leaf nodes. Although this specification limits the range of possible solutions to those that can be represented within this language, how these functions are combined together is not restricted[2].

The use of trees as a representation renders the genetic algorithm operators inapplicable, and substitute operators are used instead, although they mimic the same processes of sexual recombination and random mutation. The crossover operator works by interchanging a subtree from each of two parents, to create two offspring possessing subtrees from each of their parents. As the subtrees are not restricted to being of the same height or shape, the chromomsome trees will grow and shrink dynamically.

The mutation operator exhibits similar operational differences because of the modified representation of a chromosome. Whereas mutation in a GA operates on an individual allele, this is inappropriate for genetic programming, as the modified allele may not be of a similar type, or the same arity. Whilst mutation could be limited to interchanging functions and terminals with equivalent type specifications, this could severely limit the options available to the operator. Perhaps for this reason, the mutation operator described in (Koza, 1992)

---

[2]This statement is quantified in the case of strongly-typed functions, or problems with constrained syntactic structure, in that the representation is restricted to syntactically correct compositions.

replaces an entire randomly selected subtree with a new randomly generated one, but with a limit placed on the size of the new subtree.

Koza describes some other genetic operators specific to genetic programming. *Permutation* re-arranges the arguments of a function and is likened to the inversion operator proposed for GAs (Goldberg, 1989). *Encapsulation* recognises potentially useful sub-trees, names and stores them for later use[3]. *Editing* is a way of simplifying chromosomes that may contain superfluous functions, making them easier to display to a user but which can also be used to prune chromosomes between generations, albeit at the expense of population diversity.

As is the case with GAs, the application of the genetic operators is to those individuals selected in response to their demonstrated fitness. The concept of fitness is relatively unchanged from a genetic algorithm, except that candidate individuals are perhaps executed, rather than evaluated, to judge their performance. As is the case with genetic algorithms, the evolutionary forces acting on the population see that highly fit structures thrive and unfit structures slowly die out.

Whereas a fairly rigorous mathematical treatment of schema theory is possible for genetic algorithms, the non-linear form of chromosomes in genetic programming renders such an analysis significantly more complicated. Koza defines a schema in GP as a fully specified subtree (Koza, 1992). That is, there are no 'don't-care' symbols. Chromosomes are said to match this schema if they include this subtree somewhere within themselves. For all practical purposes, chromosomes are size bounded meaning that the set matched by a schema is finite. The average fitness of a schema in genetic programming is then defined as the average of the fitness values of all chromosomes that match the schema. Fitness-proportionate reproduction therefore acts to proliferate above-average, and eliminate below-average schemata just as for genetic algorithms. The difficulty arises with respect to calculating the probability of disruption of a schema during crossover.

For a genetic algorithm the probable disruption of a schema is calculable because it depends on the defining length of the schema and the (fixed) length of a chromosome, and is simply the quotient of schema defining length to chromosome length. Whilst the defining length of a schema in genetic programming can be viewed simply as its size (since each node has exactly one parent arc that is liable to disruption) this value is meaningless unless the total number of nodes or arcs present within a chromosome is known. As this is a dynamic value in the case of genetic programming, it is difficult to calculate the (potentially not negligible) probability of disruption. Many attempts have been made towards a schema theory for genetic programming, but these have generally concentrated on redefining the

---

[3]Such as when the mutation operator requires a new, random subtree.

genetic operators such that they have a quantifiable impact on schemata, as in (Whigham, 1995; Poli and Langdon, 1998). A schema theory that could handle the standard subtree-swapping crossover operator was finally presented in (Poli, 2001b).

## 4.2   Genetic Programming for Adapting Algorithms

Genetic programming as a general method of optimisation operates on a population of expression trees, composed of functions and terminals relevant to the problem domain. As constraint algorithms may be specified by three heuristic expressions, genetic programming is a suitable method for the adaptation of algorithms. The features identified previously that depend on the method of adaptation are foresight, hindsight and versatility.

Foresight refers to the ability of an algorithm to recognise that a heuristic which exhibits below-average independent performance may exhibit above-average performance in combination with another measure. Because genetic programming operates probabilistically based on fitness, it does not automatically exclude poor-performing candidates from future generations. Poor-performing elements will still be selected, just with reduced probability. Genetic programming is therefore able to identify synergies amongst measures.

For an adaptive system to exhibit hindsight, it must be able to learn from failure. That is, if during an iteration none of its candidate heuristics are able to solve the training instances, it is desirable that it be able to identify the candidates that made the most progress towards a solution. The fitness measure used in genetic programming allows for this, as fitness can incorporate features that do not depend on the overall resolution of an instance but on other indicative measures. Examples include the amount of the search space pruned (for a complete search procedure) or measures such as depth, mobility and coverage (Schuurmans et al., 2001) which can be used to characterise the performance of a local search. By including such measures in the fitness function, genetic programming is able to learn from failure.

Although the versatility of an adaptive system to handle both complete and local search algorithms depends on the representation employed, versatility with respect to suitability for both satisfiable and over-constrained problems depends on the method of adaptation. Genetic programming does not effect modifications that depend on information about the (explicit) internal operation or behaviour of the heuristic in question. Instead, genetic programming relies only on the fitness evaluation of the candidate heuristic, which, although obviously dependent upon the internal performance of the heuristic, is an external measure of the performance of what is otherwise a 'black-box'. Genetic programming can therefore be applied to either type of problem as it optimises based solely on the fitness measure,

which can include metrics appropriate to either satisfiable problems (such as run-time) or over-constrained problems (such as quality of solution).

## 4.2.1 An example of evolving algorithms

Having discussed how genetic programming exhibits the remaining desirable features of an adaptive constraint system, examples will now be given of heuristics posed in the representation, and of their modification with genetic programming. Consider how the well-known Novelty local search algorithm for SAT might be described as a contention heuristic in the discussed representation. As this contention function evaluates to *True* for at most one variable (move), the choice of Preference and Selection functions is not overly important. The meaning of the various functions and terminals is as follows:

`If:` The standard conditional operator.

`Not:` The standard Boolean negation operator.

`BestMinimalAge:` Returns $\top$ iff the variable in a clause that would make the most improvement in the propositional formula also has minimal age.

`IsBestInClause:` Returns $\top$ only if its variable argument is the best variable in its clausal argument.

`Is2ndBestInClause:` As before, but returns $\top$ only for the second best variable.

`StaticProbability:` Returns $\top$ a percentage of the time based on its argument P, but is static so will always return the same value in a given iteration.

`Var:` The variable under consideration (a terminal).

`StaticRandomClause:` A random clause, chosen just once each iteration (a terminal).

`P:` A numerical probability value (a terminal).

```
If(Not(BestMinimalAge(StaticRandomClause)),
    IsBestInClause(Var, StaticRandomClause),
    If(StaticProbability(P),
        Is2ndBestInClause(Var, StaticRandomClause),
        IsBestInClause(Var, StaticRandomClause)
    )
)
```

It can also be shown graphically as an expression tree (Figure 4.2).

Consider that this algorithm has been evaluated on some training instances and was sufficiently fit that it was selected to participate in recombination (cross-over). For the sake of argument, it will be crossed with another copy of itself. Cross-over requires the
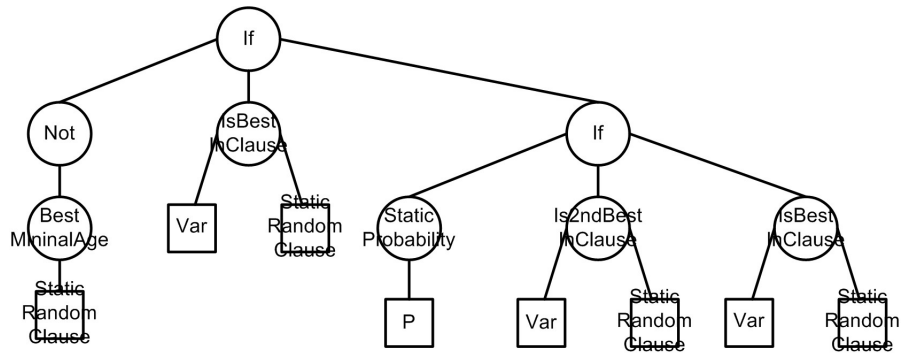
Figure 4.2: Representation of the Novelty algorithm as an expression tree.

identification of two similarly typed subtrees within the parent expressions. This is shown in Figure 4.3.

These subtrees are then interchanged between the two parent expressions, producing two new algorithms (shown in Figure 4.4). By virtue of using the same expression for both parents, and the construction of Novelty itself, the resulting child expressions are rather redundant. This is merely an artifact of the example and not something that should occur often in a diverse population. These child expressions form part of the subsequent generation allowing the process of evolution to continue.
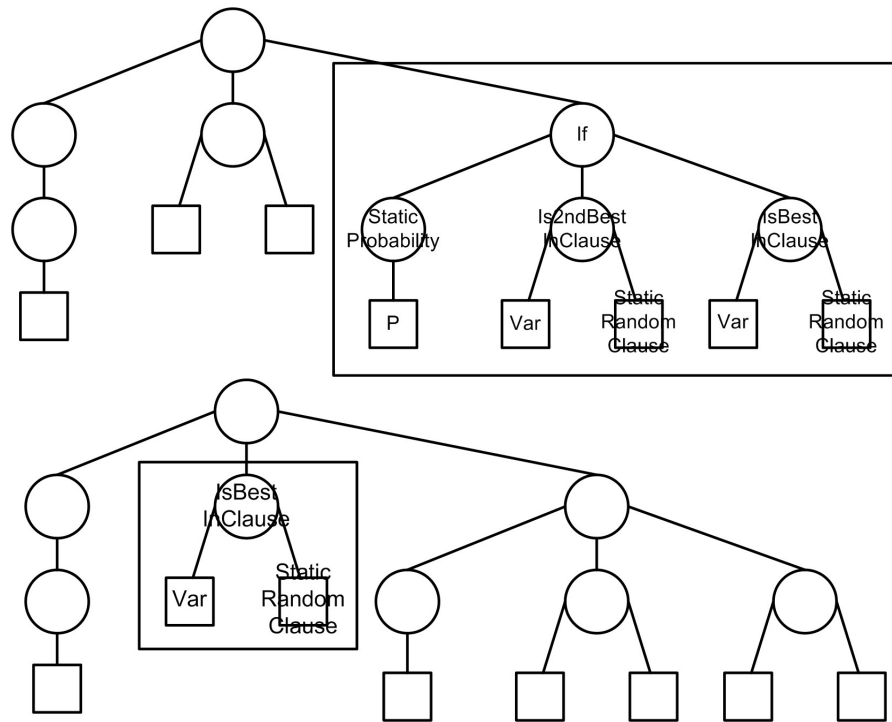
Figure 4.3: Subtrees to be extracted from the parent expressions.
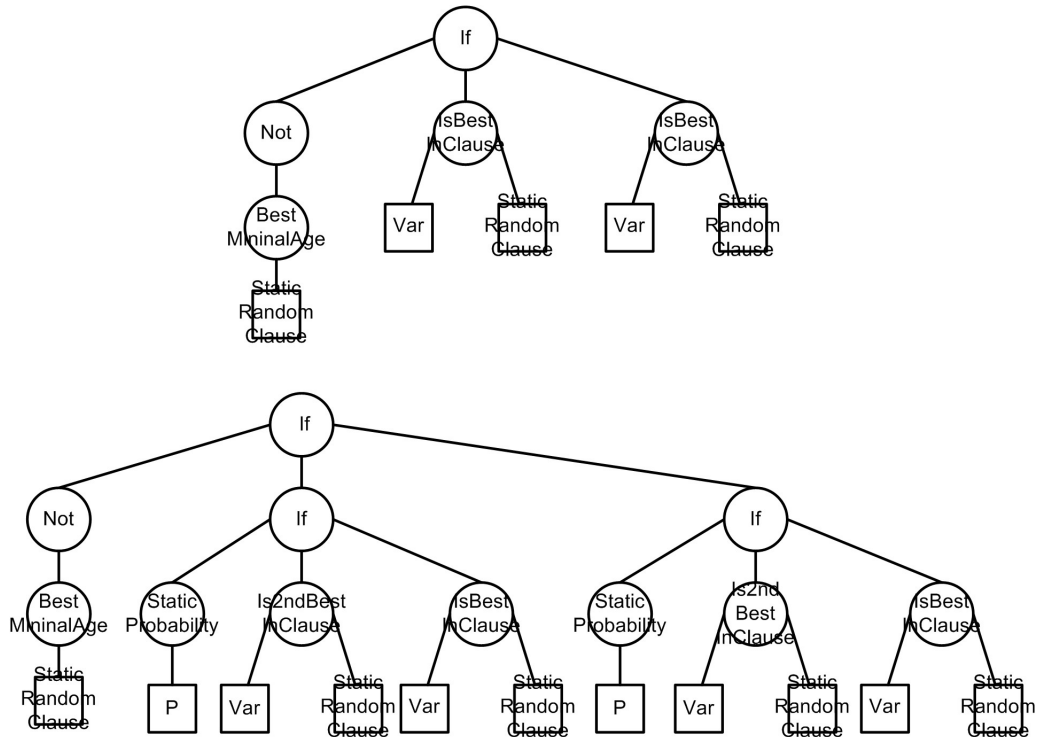


Figure 4.4: Newly generated child algorithms.

# Chapter 5

# Initial Experiments with Evolved Algorithms

This chapter introduces two independent experiments, conducted during the course of this research, examining the automated evolution of constraint algorithms. These experiments, particularly the first, were a formative influence on the identification of the five features and the development of the framework that have both been discussed in the preceding chapters.

Each work examines particular aspects of adaptive constraint systems. The first experiment (Bain et al., 2004b) examines other (non-genetic) methods of adaptation and provides supporting evidence for the choice of the latter, as well as justification for the inclusion of foresight (the ability to identify synergies) as one of the five features. The second experiment (Bain et al., 2005), although being an evaluation of the adaptive framework very much as it has been described, was intended only to demonstrate the efficacy of evolutionary methods for adaptive constraint satisfaction. It was conducted with a different experimental methodology to that which is used in the later experiments that form the main body of empirical results for this thesis. It is therefore included here for this and two further reasons: firstly, because it concerns a different problem class (over-constrained problems) than that considered in the final part of this work; but also because a number of the conclusions drawn from this study helped to shape the development of the primary empirical study of this work.

## 5.1 Methods of Automatic Algorithm Generation

In the light of a growing body of work reporting on the narrow applicability of individual heuristics, many methods of adapting algorithms to particular constraint problems have been proposed. A number of these methods have been discussed in the preceding chapter. Despite their myriad differences, all adaptive methods are motivated by the understanding that a heuristic's success on one particular problem is not an *a priori* guarantee of its effectiveness

| Functions for use in Contention Heuristics | |
|---|---|
| **InUnsatisfied ::** Move → Bool | True iff Move is in an unsatisfied constraint. |
| **WontUnsatisfy ::** Move → Bool | True iff Move won't unsatisfy any constraints. |
| **MoveNotTaken ::** Move → Bool | True iff Move hasn't been previously taken. |
| **InRandom ::** Move → Bool | True iff Move is in a persistent random constraint. The constraint is persistent this turn only. |
| **AgeOverInt ::** Move → Integer → Bool | True iff this Move hasn't been taken for Integer turns (similar to a Tabu list). |
| **RandomlyTrue ::** Integer → Bool | Randomly True Integer percent of the time. |
| **Operators for use in Contention Heuristics** | |
| **And, Or ::** Bool → Bool → Bool | The Boolean AND and OR functions. Definitions are as expected. |
| **Not ::** Bool | The Boolean NOT function which logically negates its input argument. |
| **Terminals for use in Contention Heuristics** | |
| **Move ::** Move | The Move currently being considered. |
| **NumVars ::** Integer | The number of variables in the current problem. |
| **True, False ::** Bool | The Boolean values True and False. |
| **10,25,50,75 ::** Integer | A selection of integer values. |

Table 5.1: Function and Terminal Sets for Contention

on another, structurally dissimilar problem.

Much of the research into adaptive algorithms has concerned what were described earlier as *selective* approaches, concerned with the identification of which heuristics, from a set of completely specified heuristics, are best suited for solving particular problems. This is disingenuous however, in that it assumes prior knowledge (as well as the prior existence) of the most appropriate heuristics for a given problem.

This study therefore examines three creative methods for learning new (local search) algorithms for satisfiability testing, namely a beam search, an evolutionary search and a random search. Satisfiability problems are used because they have been widely studied and have a known hardness distribution. Only a single problem instance is used for evaluation (uf100-01.cnf), taken from the uniform filtered 3-SAT benchmark set available on SATLIB (Hoos and Stützle, 2000). This problem set contains instances drawn from the phase-transition region, which is the area where the problems are (on average) the most difficult for traditional backtracking search routines.

The measures available to construct contention, preference and selection heuristics are tabulated in Tables 5.1, 5.2 and 5.3 respectively.

| Functions for use in Preference Heuristics | |
|---|---|
| **AgeOfMove ::** <br> Move → Integer | Returns the number of turns since Move was last taken. |
| **NumWillSatisfy,** <br> **NumWillUnsatisfy** <br> :: Move → Integer | Returns the number of constraints that will be satisfied or unsatisfied by Move, respectively. |
| **Degree ::** <br> Move → Integer | Degree returns the number of constraints this Move's variable participates in. |
| **PosDegree, NegDegree** <br> :: Move → Integer | Return the number of constraints satisfied by positive or negative instantiations to this variable. |
| **DependentDegree,** <br> **OppositeDegree** <br> :: Move → Integer | DependentDegree returns PosDegree if Move involves a currently True variable or NegDegree for a False variable. The reverse occurs for OppDegree. |
| **TimesTaken ::** <br> Move → Integer | Returns the number of times Move has been taken. |
| **SumTimesSat,** <br> **SumTimesUnsat ::** <br> Move → Integer | Returns the sum of the number of times all constraints affected by Move have been satisfied or unsatisfied respectively. |
| **SumConstraintAges** <br> :: Move → Integer | For all constraints Move participates in, returns the sum of the lengths of time each constraint has been unsatisfied. |
| **NumNewSatisfied,** <br> **NumNeverSatisfied** <br> :: Move → Integer | Returns the number of constraints that will be satisfied by Move that are not currently satisfied, or have never been satisfied, respectively. |
| **RandomValue ::** <br> Integer → Integer | Returns random value between 0 and Integer-1. |
| Operators for use in Preference Heuristics | |
| **Plus, Minus, Times** <br> :: Integer → Integer <br> → Integer | Returns the arithmentic result of its two integer arguments. |
| **LeftShift** <br> :: Integer → Integer | Returns its input shifted 16 bits higher. |
| Terminals for use in Contention Heuristics | |
| **Move ::** Move | The Move currently being considered. |
| **NumVariables,** <br> **NumConstraints** <br> :: Integer | The number of variables and constraints in the current problem. |
| **NumFlips ::** Integer | The number of Moves that have already been made. |
| **0, 1 ::** Integer | The integers 0 and 1. |

Table 5.2: Function and Terminal Sets for Preference

| Functions for use in Selection Heuristics | |
|---|---|
| **RandomFromMax,** **RandomFromMin,** **RandomFromPositive,** **RandomFromAll ::** CostList → Move | The first two functions make a random selection from the maximum or minimum cost moves, respectively. The third makes a random selection from all moves with a positive preference value. The final function makes a random selection from all moves in the preference list. |
| Terminals for use in Selection Heuristics | |
| **ListOfCosts ::** CostList | The list of costs determined by the preference stage. |

Table 5.3: Function and Terminal Sets for Selection

## 5.1.1   Beam Search

Beam search is an effective method of controlling the combinatorial explosion that can occur during a breadth first search. It is similar to a breadth first search, but only the most promising nodes at each level of search are expanded. The primary limitation of beam search is its inability to recognise and exploit synergies that may exist in the problem domain. Whether such synergies exist within the domain of heuristics, and therefore whether their recognition is an important feature of an adaptive system, is an empirical question that is addressed in this experiment.

To determine whether such synergies occur, a study of possible contention heuristics was conducted using a beam search. The $i$-th ply of the search tree includes all possible heuristics that can be enumerated using exactly $i$ functional measures from Table 5.1. As contention heuristics are Boolean functions, the compound heuristics in deeper plies ($i > 1$) are formed by combining functional measures using the Boolean operators AND, OR and NOT. As contention heuristics cannot be considered in isolation from preference and selection heuristics, the heuristics adopted for these stages were the preference and selection heuristics of the GSAT local search algorithm (Selman et al., 1992), being the selection of the candidate move that will satisfy the maximum number of clauses.

The available contention measures provide for 16 algorithms in the first iteration of the beam search. The specification of the contention heuristic of each is given in the left hand side of Table 5.4. The heuristics are ranked by their percentage success rate, which is given in the table along with the average number of flips required to determine that the problem instance is satisfiable. Algorithm runs that had failed to demonstrate the instance satisfiable within 40,000 moves were aborted. Each algorithm was allowed 500 independent tries on the problem instance, but restarts were not used within each try. Unsuccessful runs are included as part of the calculation of the average number of moves to solution. The results suggest a natural delineation of the heuristics (measures) into two groups: those that were generally able to solve the problem instance (heuristics ranked 1-6) and those that were not (which

are ranked 7-16).

| Problem: uf100-01, Tries: 500, Cutoff: 40000 | | | | | | | |
|---|---|---|---|---|---|---|---|
| Heuristics with up to one functional node | | | Beam search up to two functional nodes | | | | |
| Rank | Algorithm | (% Solved) Avg. Flips | Beam Width | Domain Size | Best Avg. Flips | Percent Improv. | Best % Solved |
| 1 | AgeOverInt(Move, 10) | (76%) 21924 | | | | | |
| 2 | RandomlyTrue(50) | (71%) 20378 | 2 | 4 | 20105 | 1.34 | 69% |
| 3 | RandomlyTrue(25) | (67%) 23914 | 3 | 9 | 11262 | 44.73 | 98% |
| 4 | RandomlyTrue(75) | (50%) 24444 | 4 | 16 | 11262 | 44.73 | 98% |
| 5 | True | (36%) 28111 | | | | | |
| 6 | RandomlyTrue (NumVariables) | (35%) 28846 | 6 | 25 | 11262 | 44.73 | 98% |
| 7 | InUnsatisfied(Move) | (1%) 39455 | 7 | 36 | 1988 | 90.24 | 100% |
| 8 | AgeOverInt(Move, 25) | (1%) 39893 | | | : | | |
| 9 | RandomlyTrue(10) | (0%) 39936 | | | : | | |
| 10 | False | (0%) 40000 | | | : | | |
| 11 | AgeOverInt(Move, 75) | (0%) 40000 | | | : | | |
| 12 | AgeOverInt(Move, 50) | (0%) 40000 | | No further improvement | | | |
| 13 | AgeOverInt(Move, NumVariables) | (0%) 40000 | | | : | | |
| 14 | InRandom(Move) | (0%) 40000 | | | : | | |
| 15 | MoveNotTake(Move) | (0%) 40000 | | | : | | |
| 16 | WontUnsatisfy(Move) | (0%) 40000 | 16 | 196 | 1988 | 90.24 | 100% |

Table 5.4: Beam Search Results

The second iteration of the beam search allows heuristics to use (at most two) of the $B$ best measures from the first iteration. Various settings of $B$ were considered, ranging from 2 to 16. The 4 heuristics possible when $B = 2$ are: AgeOverInt(Move, 10) AND RandomlyTrue(50); AgeOverInt(Move, 10) OR RandomlyTrue(50); NOT AgeOverInt(Move, 10); and NOT RandomlyTrue(50) (duplicate heuristics due to the commutativity of AND and OR were ignored). None of these four heuristics offered a significant improvement over the simpler heuristics, however. A significant improvement in heuristic performance is observed with $B = 3$, which allows RandomlyTrue(25) to also form part of candidate heuristics. Although the inclusion of this measure raised the success rate of the best candidate heuristic to almost 100%, there was no further improvement in heuristic performance for any beam width setting within the range of the prior 'good' heuristics.

Only by increasing the beam width to allow the inclusion of a measure that exhibited particulary poor independent performance (InUnsatisfied(Move)) does an improved heuristic result. Furthermore, the improvement resulting from the inclusion of this measure is drastic, reducing the average run-length by more than a factor of five. Although the efficacy of this measure is obvious to human developers (most local search algorithms include it), it is not at all obvious to beam search, where its poor individual performance denotes it as a measure to be considered later, if at all.

In this case, a beam width of only $B = 7$ was required to realise the best heuristic that can be created using at most two measures. Although the 36 algorithms that this allows is not excessive, this number is reduced by the commutativity of the Boolean operators, and would be much larger in the general case. That only the best of the poorer measures was required to achieve this performance is merely fortuitous. As there was so little difference between any of the poorer measures, considering any of them strongly supports trying them all.

This experiment has demonstrated that within a potential domain of contention measures, synergies do occur between measures. Synergies were also observed to occur between measures that exhibited particularly bad independent performance. Hence, this domain does not satisfy the necessary requirements for the successful application of a beam search procedure, which is that measures exhibiting poor independent performance will not lead to improved performance and are therefore not required to be considered further.

Whilst it may be possible to locate good heuristics using a beam search, the width of the beam necessary to account for the poor independent performance of genuinely effective measures eliminates much of the computational advantage of the method.

## 5.1.2   Evolutionary Exploration of the Seach Space

Genetic programming (Koza, 1992) has been proposed for discovering solutions to problems when the form of the solution is not known. Genetic programming potentially resolves two of the limitations identified in existing work, namely the inability to exploit synergies and the inability to learn from failure. Synergies can be exploited as individuals are selected probabilistically to participate in reproduction, rather than deterministically as occurs in beam search. This allows individuals exhibiting poor independent performance the (albeit reduced) possibility of comprising part of a subsequent generation. And although not a limitation of beam search, genetic programming is also able to learn from failure, as the fitness function can comprise much more information than just whether or not a solution was found.

The performance of genetic programming (and genetic algorithms) is often attributed to schema theory, meaning that it is above-average schema, and not individuals, that proliferate within a population. Assuming that schema theory is correct, it is imperative that above-average heuristic schemas be able to proliferate within a population of heuristics. For this to occur requires that: 1) above-average schemas must exist (at least for the problem domain of interest) 2) furthermore, they must offer sufficiently improved performance that they are selected frequently enough to overcome the probability of disruption, which is also to say that

3) they must not be excessively large. This experiment will test whether these conditions hold by empirically evaluating the performance of a genetic programming experiment for evolving algorithms.

As well as combining different contention, preference and selection heuristics in novel ways, the inclusion of functions like AND, OR, PLUS and MINUS permit a range of new heuristics to be learned. As genetic programming is being permitted to learn new preference and selection heuristics, whereas the earlier beam search experiment was not, it is not intended to directly compare the performance of genetic programming to the earlier beam search. This experiment serves only to test the efficacy of genetic programming within this domain.

Parameters and results of the experiment can be found in Table 5.5. Results are not monotonically improving due to the non-determinism of the learned local search heuristics. These results show that the genetic programming method rapidly evolves efficient algorithms from an initially poor performing random population. The mean results refer to population means, but the best performing algorithms of later generations achieved a 100% success rate on the test problem. That the procedure was not able to locate a heuristic that performed as well as the state of the art is perhaps best explained by the fact that the available set of preference measures did not include the SAPS weighting heuristics. Although the experiment was continued for 100 generations, there was little improvement after generation 30. The performance of the heuristics learned with the evolutionary procedure is plotted against generation in Figure 5.1.

| Experiment Conditions | | Experimental Results | | | | |
|---|---|---|---|---|---|---|
| **Population Composition** | | Gen. | Mean Success | Mean Unsat. | Best Avg. Moves | Best So Far |
| Population Size | 100 | | | | | |
| Elitist copy from previous gen. | 25 | 0 | 0.04% | 34.89 | 38435 | 38435 |
| Randomly selected and crossed | 70 | 10 | 9.52% | 13.45 | 9423 | 9423 |
| New elements generated | 5 | 20 | 65.68% | 3.16 | 1247 | 1247 |
| **Evaluation of Algorithm Fitness** | | 30 | 83.23% | 2.35 | 981 | 981 |
| $\boldsymbol{F}_i{=}Standardised(UnsatConstraints_i)+$ | | 40 | 85.12% | 3.04 | 1120 | 981 |
| $100 * SuccessRate_i$ | | 50 | 89.88% | 3.14 | 1131 | 981 |
| Test Problem | uf100-01 | 60 | 91.96% | 2.15 | 898 | 898 |
| Number of runs for each algorithm | 25 | 70 | 88.96% | 1.90 | 958 | 898 |
| Maximum moves per run | 40000 | 80 | 89.04% | 2.64 | 1062 | 898 |
| Mean moves required by the state- | | 90 | 90.56% | 1.35 | 876 | 876 |
| of-the-art (Hutter et al., 2002) | 594 | 99 | 92.88% | 1.73 | 1070 | 876 |

Table 5.5: Conditions and Results for the Genetic Programming Experiment

As genetic programming has been shown to improve a population of heuristics, this suggests that the domain of heuristic schemas satisfies all three requirements outlined earlier.
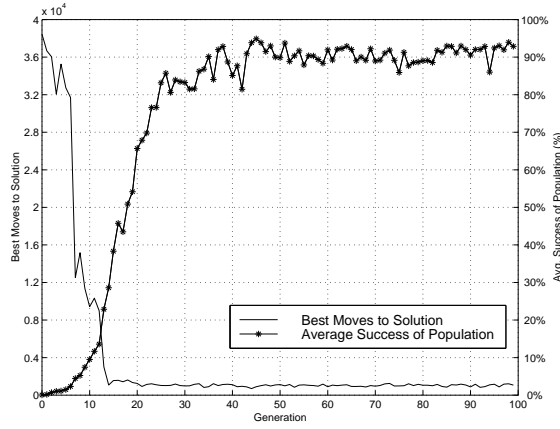
Figure 5.1: Results of the genetic programming experiment

What this experiment has not determined is whether the genetic operators are truly responsible for the observed improvement in the population.  The hypothesis that it is not the genetic operators, but merely coincidental random improvements persisting through the operation of elitist cloning, is tested in the following section.

### 5.1.3   Random Exploration of the Search Space

In order to demonstrate that the observed performance improvements in the evolutionary experiment are attributable to the genetic operators and not simply the result of fortuitously random interactions, the preceding experiment was repeated without the genetic operators. That is, each generation of the population was composed entirely of randomly generated elements, with the performance of the best element in each generation recorded.  As genetic programming must begin with a similar randomly generated population, any observed differences in overall performance between the random experiment and the evolutionary experiment, can be attributed to the genetic operators of selection, cross-over and cloning.

With the exception of the differences in population composition, parameters for this experiment were the same as for the previous experiment. Results are shown in Table 5.6, when three different (practical) limits are placed on the size of the generated contention and preference trees[1]. Only the best average moves to solution (so far) and the best success rate (so far) are reported, as generational averages have no meaning within the context of this experiment.

The results clearly show that a random exploration of the search space does not approach the performance of an evolutionary method.  From this it can be concluded that the use

---

[1]Selection heuristics are already restricted by the function and terminals sets to have exactly 2 nodes.

| | Node Limit = 6 | | Node Limit = 20 | | Node Limit = 80 | |
|---|---|---|---|---|---|---|
| Gen. | Best Average Moves | Best Success % | Best Average Moves | Best Success % | Best Average Moves | Best Success % |
| 0 | 33981 | 32 | 38424 | 4 | 40000 | 0 |
| 10 | 33543 | 32 | 33531 | 20 | 23671 | 64 |
| 20 | 33543 | 32 | 6301 | 100 | 23671 | 64 |
| 30 | 6959 | 92 | 6301 | 100 | 23671 | 64 |
| 40 | 6959 | 92 | 6301 | 100 | 23671 | 64 |
| 50 | 6959 | 92 | 6301 | 100 | 23671 | 64 |
| 60 | 6959 | 92 | 6301 | 100 | 20814 | 88 |
| 70 | 6959 | 92 | 6301 | 100 | 6726 | 100 |
| 100 | ... | | No further improvement | | ... | |

Table 5.6: Results for the Random Exploration Experiment

of fitness and the genetic operators are responsible for the performance observed in the preceding experiment.

## 5.2 Evolving Heuristics for Constrained Optimisation

The previous section reported a study confirming the efficacy of genetic programming in adapting heuristics for solving (a particular) propositional satisfiability problems. As learning a heuristic appropriate for just a single problem instance is not overly useful, this next study examines how heuristics can be evolved that are effective on broader classes of problems.

The problem classes to be examined are still propositional satisfiability problems. However, as all problem instances considered are over-constrained they are in effect MAX-SAT problems. Rather than demonstrating satisfiability, the goal (at least in this study) is to prove optimality of a solution.

Algorithms to solve MAX-SAT problems encounter a number of additional challenges absent when determining the satisfiability of a constraint problem. Firstly, a local search is unable to recognise the optimality of a solution, unless the optimal cost has already been determined using some other method. Secondly, for a complete search to guarantee optimality, the search space of a MAX-SAT problem must be thoroughly examined (in contrast to that of a satisfiable problem where execution may terminate once a satisfying solution has been found). Additionally, until the current cost bound is exceeded, backtracking search must overlook constraint violations that would have triggered immediate backtracking in satisfiability testing.

To overcome these challenges, recent work (Borchers and Furman, 1999; Xing and Zhang, 2004) has adopted a two-phase approach that firstly uses a greedy local search routine to quickly locate near-optimal solutions. The best of these solutions then sets the ini-

tial bound for a branch and bound procedure, which determines the globally optimal solution. The branch and bound approach used in (Borchers and Furman, 1999) relies on unit clause propagation and a dynamic variable ordering determined by the well-known MOMS heuristic, which selects the variable exhibiting the *M*aximum *O*ccurrences in clauses of the *M*inimum *S*ize, for instantiation. More recently, the *MaxSolver* ordering heuristic (Xing and Zhang, 2004) was applied to MAX-SAT, which gives additional consideration to the clause-to-variable ratio of the problem. This, in conjunction with a collection of other variable propagation rules, achieved performance superior to that of MOMS with unit-propagation.

Each of these works relies on a single ordering heuristic that has been demonstrated to perform well on a generalised range of benchmark instances. However, good performance on such instances is not necessarily indicative of superior performance on other specific problems, i.e. generality comes at the expense of specific efficiency. The same limitation is known to apply more formally to local search methods, with the no-free-lunch theorems (Wolpert and Macready, 1997) determining that a heuristic algorithm's performance, averaged over the set of all possible problems, is identical to that of any other algorithm.

To overcome this limitation of a single heuristic approach, this study examines the evolution of complete (backtracking) search heuristics. Evolved algorithms are shown to significantly outperform existing approaches on a range of NP-hard MAX-SAT problems.

## 5.2.1   Methodology

Although adaptation of all three heuristic functions for the contention, preference and selection stages of search can occur simultaneously, an *a priori* commitment to one or more of these can be made instead. This study is concerned only with the adaptation of preference heuristics for a branch and bound procedure. A natural choice for the contention function is the most general one possible for a complete search: contend all moves for currently unvalued variables. Similarly appropriate to this context is the selection function: select the move with the highest preference value (breaking ties by preferring the variable lowest in the chronological ordering).

The commitment to these two functions leaves the system to evolve preference functions that are to be maximised for the perceived best move. Although it may appear that a commitment to maximise the preference function is restrictive, in that it does not allow the system to recognise that certain measures should be minimised, this is not in fact the case. The presence of negative valued constants allows the system to convert each measure into its corresponding anti-measure simply by multiplication. Tie-breaking heuristics are also realisable, as it is possible to multiply a measure by a constant large enough so that it

always dominates an added secondary, tie-breaking measure. The measures and constants available for constructing preference functions are given in Table 5.7.

An initial evaluation of the Table 5.7 heuristics revealed that MOMS exhibited the best independent performance on a range of problems. For this reason, all evolved Preference functions are constrained by requiring them to incorporate MOMS, as shown in the expression tree of Figure 5.2. Although the evolutionary procedure should be able to independently identify this fact, it is performed for the practical consideration that many heuristics will perform very poorly independently, and there can be little point in evaluating numerous poorly-performing algorithms. As both of the MAX-SAT approaches discussed earlier (Borchers and Furman, 1999; Xing and Zhang, 2004) used MOMS or a variation of it, it is not unreasonable that the evolutionary procedure begins with, and attempts to improve upon, the current approach.

Although nodes 0 through 2 are protected from modification, this constraint does not in any way limit the generality of the heuristics that may be learned, or affect the diversity of the population. Diversity is maintained by the unconstrained subtree rooted at node 3, whilst generality is preserved since the system can counteract the protected heuristic by incorporating $Times(MOMS(move), -1)$ somewhere into the unconstrained subtree. The underlying branch and bound algorithm, in which the learned heuristics are used, additionally makes use of unit-propagation rules UP1-UP3, as presented in (Xing and Zhang, 2004) (but these should be viewed as a meta-level activity of algorithm).
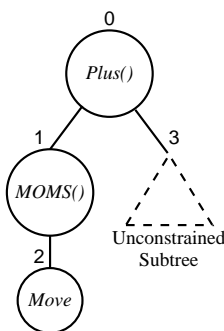


Figure 5.2: Base algorithm expression tree.

Apart from the initial random generation, the composition of each subsequent population is determined by the fitness of individual algorithms. The fitness measure in these experiments was the number of backtracks necessary to determine the optimal solution to the training instance, standardised by the observed maximum so that fewer backtracks equates to higher fitness. Each candidate algorithm was allowed the same number of backtracks required by the MOMS algorithm for that particular instance (rounded up to the nearest

| Measure :: Type | Description |
|---|---|
| **Linear ::** Move $\Rightarrow$ Float | Maximal for chronologically first, uninstantiated var. |
| **FwdDegree, BwdDegree** **Degree ::** Move $\Rightarrow$ Float | For a *move*'s variable, the number of uninstantiated, instantiated or total neighbours respectively. |
| **NumConstraints ::** Move $\Rightarrow$ Float | Number of constraints that this *move*'s variable participates in. |
| **Determined,** **Undetermined** Move $\Rightarrow$ Float | Determined clauses are those that are satisfied, or are unsatisfied with no uninstantiated variables remaining. These heuristics count the un/determined clauses for a *move*'s variable. |
| **MOMSStrict ::** Move $\Rightarrow$ Float | Counts the number of occurrences in clauses of minimum size for a *move*'s variable. |
| **MOMSLiteral ::** Move $\Rightarrow$ Float | As above, but considers the actual *move* rather than the *move*'s variable. |
| **MOMS ::** Move $\Rightarrow$ Float | Sum of weighter clause size: $M(move) + M(\neg move)$ |
| **2SidedJW ::** Move $\Rightarrow$ Float | The 2-sided Jeroslow-Wang rule: $J(move) + J(\neg move)$ |
| **JeroslowWang,** **RevJW ::** Move $\Rightarrow$ Float | The Jeroslow-Wang rule: $J(move)$ and the reverse Jeroslow-Wang rule: $J(\neg move)$ respectively. |
| **1stOrder ::** Move $\Rightarrow$ Float | A 1st-order approximation to the Jeroslow-Wang rule: $J(move) - J(\neg move)$ |
| **ValUsed ::** Move $\Rightarrow$ Float | Number of times a *move*'s value previously tried. |
| **CountSatisfy ::** Move $\Rightarrow$ Float | Evaluates the total weight of clauses that will be become satisfied by a *move*. |
| **UndetCount ::** Move $\Rightarrow$ Float | The total count of undetermined clauses a *move*'s variable is involved in. |
| **NumWillDetermine ::** Move $\Rightarrow$ Float | The number of clauses that will be determined by the *move*. |
| **UnitClause ::** Move $\Rightarrow$ Float | Counts undetermined clauses that will become unit clauses after this *move*. |
| **Plus ::** Float $\rightarrow$ Float $\Rightarrow$ Float | Calculates the sum of its two input arguments. |
| **Times ::** Float $\rightarrow$ Integer $\Rightarrow$ Float | Calculates the product of a floating point and integer input arguments. |
| **Times2 ::** Float $\rightarrow$ Float $\Rightarrow$ Float | Calculates the product of its two floating point input arguments. |
| **Move ::** Move | A *move*, i.e. the assignment of True or False to a var. |
| **[-10..-1, 1..10] ::** Integer | Integer constants for changing heuristic importance. |

Table 5.7: Measures and constants for use in Preference functions. $J(x) = \sum_{c \in C_x} 2^{-n_c}, M(x) = \sum_{c \in C_x} 5^{-n_c}$. $C_x$ is the set of undetermined clauses containing literal $x$, $n_c$ is the number of uninstantiated literals in clause $c$. All measures accepting a *move* are scaled by that measure's current maximum, to return a value in the range $[-1..1]$.

1024). In cases where evaluation did not complete within this limit, the estimated number of backtracks was calculated by multiplying the number of backtracks that had occurred by the percent of the search space the algorithm had eliminated at the time when evaluation ceased. Although this fitness function favours algorithms that quickly prune large parts of the search space, the algorithms of highest fitness remain those that completed their evaluation within the stated limit.

Each generation consisted of $n_p = 50$ algorithms, with elitism ensuring that the $n_c = 3$ fittest algorithms are not lost by directly copying those elements to the succeeding generation. This fitness measure also determines the selection of $n_b = 36$ algorithms that will be used to breed the next generation. Selection is fitness proportionate, but not random, so an individual $i$ with fitness $f_i$ is guaranteed to be selected at least $\lfloor n_b f_i / \sum_{j=1}^{n_p} f_j \rfloor$ times. Random pairs of algorithms from this selection are crossed-over by randomly interchanging a subtree of each, creating $n_b$ new algorithms. The selected subtrees must be essentially different, but have the same return type. In cases where crossover fails to generate new trees, the selected individual trees are mutated, as explained below.

Mutation in (Koza, 1992) is a fairly drastic procedure, replacing a subtree with an entirely new, randomly generated one. To preserve the fitness of the population and to encourage the exploration of synergies, this type of mutation is used with only 33% probability. At other times, the existing subtree is maintained but becomes the left subtree of the $Plus()$ heuristic (33% chance) or the $Times()$ heuristic (33% chance). The corresponding right subtree is randomly generated. The subtree selected for mutation is initially the subtree rooted at node 3, and only if this procedure fails is a different subtree selected. The best $n_m = 11$ elements are selected for mutation. These parameter values were selected simply by trial and error, so a more thorough tuning of these parameters may lead to additional performance gains.

Four different classes of problems were selected from which training instances were drawn. These were hard random MAX-3-SAT problems (*uuf100*) from SATLIB; unsatisfiable *jnh* problems from the DIMACS benchmark set; random MAX-2-SAT problems from Borchers' work[2] (Borchers and Furman, 1999); and SAT encoded unsatisfiable quasigroup instances, generated according to the method and descriptions in (Zhang and Stickel, 2000).

Each problem instance is used as the sole training instance for an evolutionary experiment, and was repeated 5 times for each training instance. The evolutionary procedures were run for 50 generations in all cases, with a practical limit of 20 nodes placed on the size of each tree. The best of the 5 independent experiments is accepted as the evolved

---

[2]For clarity, instances are named as p_*CLAUSESIZE_#VARS_#CONSTRAINTS*.

heuristic for that training instance. The performance of the evolved heuristic from each training instance is then evaluated on a larger class of problem instances of the same class, and compared against a standard MOMS variable ordering heuristic, and also *MaxSolver* (Xing and Zhang, 2004)[3]

## 5.2.2 Training Results

The evolution of two different types of heuristics was considered, which can be described as linear combinations and non-linear combinations. Linear combinations do not allow the multiplication of measures with one another, only with numeric constants, so that the learned algorithms are a weighted sum of the different measures. Conversely, non-linear combinations do allow the multiplication of measures together. Practically, this is achieved by the presence of two multiplication operators, one for each type of multiplication. As the representation is strongly typed, removal of the $Times2()$ operator implicitly disallows any non-linear (multiplicative) combinations of measures, restricting the space of possible heuristics to include only weighted sums of the various measures. This separation was motivated by the fact that some existing adaptive systems, specifically ACE (Epstein et al., 2002), allow only a limited number of non-linear combinations, and this separation allows the efficacy of more complicated non-linear combinations to be examined.

The specific training instances are listed in Table 5.8, and comprised (with respect to their difficulty for a MOMS ordering heuristic) easy (but non-trivial) MAX-2-SAT problems, the most difficult *jnh* and *uuf100* instances[4], and an order 5 quasigroup problem. Time and backtrack comparisons with MOMS and the expression for the evolved algorithm requiring the fewest backtracks are also shown.

Every evolved algorithm required fewer backtracks than MOMS on its training instance. In terms of the backtracks as a percentage of MOMS backtracks, the linear algorithms offered mean and median improvements of 56.2% and 62.45% respectively. The algorithm offering the least improvement still required 16.0% fewer backtracks than MOMS on its training instance. The most significant improvements were observed with the algorithms evolved for the *uuf100* problems, the best requiring only 4.2% of the backtracks required by MOMS.

The presence of non-linear combinations resulted in heuristics that outperformed their linear counterparts on 10 of the 14 training instances (the performance of the best algorithm for each instance is shown in bold), results for which have also been included in Table 5.8.

---

[3]It should be noted that MaxSolver employs an additional unit-propagation procedure for 2-SAT clauses not implemented in our system, and which is therefore prejudicial against our results. Furthermore, some problem classes (*jnh* and quasigroup) could not be evaluated with *MaxSolver* because the released version does not support problems with greater than 100 variables of 500 clauses.

[4]The most difficult instances were selected from these two classes because the simpler instances are trivial.

| Instance | Cost (GSAT / Optimal) | MOMS BTs | Evolved Linear BTs | Time %MOMS | With Non-linear BTs | Time %MOMS |
|---|---|---|---|---|---|---|
| uuf100-0420.cnf | (2/2) | 11030 | **8571** | 123.8% | 8580 | 134.3% |
| MOMS+Degree-3*(RevJW+Linear) | | | | | | |
| uuf100-04.cnf | (2/2) | 11085 | **8491** | 132.0% | 8706 | 116.9% |
| MOMS+CountSatisfy+10*(ValUsed-10*FwdDegree) | | | | | | |
| uuf100-0327.cnf | (3/1) | 17950 | 748 | 8.64% | **472** | 4.3% |
| MOMS+MOMSLiteral*(2SJW-NumWillDetermine) | | | | | | |
| uuf100-0190.cnf | (3/2) | 31064 | 7524 | 42.5% | **5969** | 35.2% |
| MOMS+2SidedJW*(Linear*MOMSStrict+1)+MOMSStrict | | | | | | |
| uuf100-0332.cnf | (3/2) | 46709 | 6015 | 25.5% | **5378** | 23.1% |
| MOMS+1stOrder+MOMS*(MOMSStrict+NumWillDetermine+RevJW) | | | | | | |
| p_2_50_200.cnf | (16/16) | 5835 | **783** | 23.2% | 798 | 21.6% |
| MOMS+20*(100*FwdDegree+2SidedJW) | | | | | | |
| p_2_50_250.cnf | (22/22) | 27610 | 5827 | 35.5% | **4855** | 31.8% |
| MOMS+(Undetermined+UnitClause)*(JeroslowWang*1stOrder*FwdDegree) | | | | | | |
| p_2_100_300.cnf | (15/15) | 84062 | 6619 | 15.4% | **6521** | 14.1% |
| MOMS+(72*Undetermined$^2$*FwdDegree*Degree) | | | | | | |
| jnh310.cnf | (3/3) | 4744 | 3923 | 89.2% | **3711** | 122.4% |
| MOMS+Degree+(UndetCount*FwdDegree*MOMSStrict*NumConstraints) | | | | | | |
| jnh307.cnf | (3/3) | 5244 | **2668** | 68.1% | 3177 | 100.0% |
| MOMS+FwdDegree-180*Determined | | | | | | |
| jnh303.cnf | (3/3) | 21554 | 18102 | 132.3% | **15830** | 103.9% |
| MOMS+BwdDegree+(BwdDegree*MOMSLiteral*UnitClause*Linear) | | | | | | |
| jnh302.cnf | (4/4) | 35335 | 29458 | 122.7% | **25440** | 137.1% |
| MOMS+MOMSStrict*(UnitClause*UndetCount*MOMSLiteral+1) | | | | | | |
| jnh305.cnf | (4/3) | 39104 | 7984 | 29.5% | **7498** | 37.3% |
| MOMS+CountSatisfy*(MOMSStrict-1stOrder*MOMSLiteral) | | | | | | |
| qg3-05.cnf | (5/5) | 21935 | 11817 | 79.1% | **10998** | 71.3% |
| MOMS+(JeroslowWang*UnitClause+ValUsed)*(MOMSStrict*Linear) | | | | | | |

Table 5.8: Comparison of performance of evolved algorithms on training instances, along with the preference expression of the best performing algorithm for each. Boldface denotes the algorithm requiring the fewest backtracks.

The mean and median improvements of the non-linear variants over MOMS were 58.4% and 65.32%, both greater than the respective figures for linear combinations. From these results it can be seen that more complex combinations of heuristics are advantageous.

Although the evolved heuristics are clearly better than MOMS in terms of backtracks, there is less distinction in terms of time performance, with MOMS exhibiting better time performance for 4 of the 14 instances. This is mostly explained by our implementation needing to represent any combination of heuristics, and therefore acts more like an interpreted language than a compiled one. Specialised and optimised data structures that would have improved the time performance of an individual algorithm were therefore not possible. This should not be taken as a weakness of the representation or of genetic programming, but simply as an implementation constraint. For a production situation, the evolved algorithms would be implemented using only the necessary data structures and operations.

### 5.2.3   Testing results

If an evolutionary method of search is suited to adapting algorithms, then for each particular training instance it should have identified the best algorithm for that instance. Since the best algorithm for each instance is unknown (and potentially unknowable), it is possible only to say that of all the known algorithms, each algorithm should be the best for its particular instance. To test this, a cross-validation was performed where each evolved algorithm (both linear and non-linear) was evaluated on all 14 training instances. When compared with all other evolved algorithms, each was found to be the best for its particular instance, demonstrating that an evolutionary method of search is locating the 'best' algorithm for each particular problem instances.

Due to the computational time required for training, the ability of a heuristic to perform well on a single training problem is not particularly useful. Instead, good performance on a training instance must translate into good performance on a class of similar problems. To demonstrate that the evolved heuristics exhibit such performance, the linear and non-linear evolved algorithms for each training instance were all evaluated on a larger test set of problems of their class. The test set comprises: for MAX-3-SAT, the first 100 *uuf100* instances; 6 MAX-2-SAT problems (training instances, *p_2_50_300*, *p_2_50_350* and *p_2_100_400*); all 34 unsatisfiable *jnh* instances; and quasigroup instances of order 5, 6 and 7. Two test sets (*uuf100* and *jnh*) contain problems of similar size and structure, whilst the remaining two sets contain instances of similar structure but larger than the original training instances. Performance results for MOMS and the best evolved algorithm in each class are tabulated in Table 5.9. These results and those for *MaxSolver* are also plotted as cumulative run-lengths

| Class | Num. | MOMS BTs | | Linear BTs | | Non-Linear BTs | |
|---|---|---|---|---|---|---|---|
| | Insts | Mean | Median | Mean | Median | Mean | Median |
| *uuf100* | 100 | 1311 | 295 | 1308 | 288 | 1320 | 289 |
| *jnh* | 34 | 3592 | 550 | 2665 | 457 | 2325 | 475 |
| *MAX-2-SAT* | 6 | 3.16E6 | 629613 | 410180 | 87362 | 100070 | 34628 |
| *quasigroup* | 3 | 2.95E6 | 311986 | 238950 | 132901 | 69037 | 84530 |

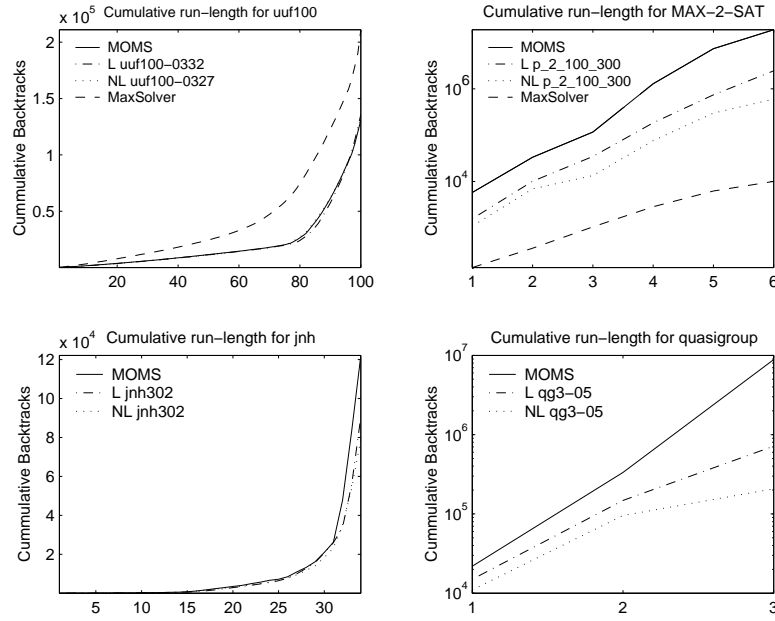Table 5.9: Evolved algorithm performance on test sets



Figure 5.3: Evolved Algorithms vs MOMS Cumulative Run-Lengths. X-axes indicate the number of instances in the class.

in Figure 5.3.

Although an evolved algorithm was able to outperform *MaxSolver* on the *uuf100* test set, there was no discernible improvement over a generic MOMS ordering heuristic. This is hardly surprising, given that this is such a large, random problem set. Using just a single training instance is perhaps insufficient to truly characterise the class, even if such a random problem class does exhibit some structure that a specialised algorithm might be able to exploit. On every other problem class however, an evolved algorithm required significantly fewer backtracks than MOMS. Furthermore, it can be seen from Figure 5.3 that the evolved algorithms dominate MOMS on both the quasigroup and MAX-2-SAT sets, requiring fewer cumulative backtracks over the entire test set. In particular, the non-linear variants can be seen to outperform the linear variants on all but the *uuf100* set, whilst maintaining

similar performance on that class. *MaxSolver* did outperform an evolved algorithm on the MAX-2-SAT set, but this is partly attributable to the additional unit-propagation rules that *MaxSolver* applies to MAX-2-SAT problems.

A Wilcoxon Rank-Sum test (also known as the Mann-Whitney U-Test) was performed to further compare the evolved algorithms to MOMS. The Wilcoxon test is a non-parametric method used for identifying whether two (not necessarily normal) distributions are of the same form, but shifted. As algorithm run-times and run-lengths often exhibit heavy-tailed behaviour, this test is ideally suited for comparing the performance of constraint algorithms. On every class, the Wilcoxon test revealed that an evolved algorithm should be preferred, exhibiting a smaller rank-sum than MOMS, but was unable to establish statistical significance with a confidence interval of 5%. In part, this can be attributed to the small number of instances in two of the test sets.

Furthermore, the Wilcoxon test revealed that non-linear algorithms had equal or smaller rank-sums than the linear variants on all four problem classes considered. Although exhibiting only comparable performance on the *uuf100* test set, non-linear variants achieved superior performance on all other test sets. This suggests that non-linear combinations identify and exploit synergies overlooked by a linear approach, and whilst not appropriate for all problem classes, there can be substantial performance gains from non-linear combinations on problem classes that are sufficiently homogenous.

## 5.3   Conclusions

These two studies demonstrated the ability of an evolutionary method to learn improved algorithms for specific constraint satisfaction instances. In the second study, these evolved algorithms were then evaluated on a larger set of test problems drawn from the same class, and were shown to outperform existing approaches in a number of instances. However, a number of improvements to the experimental methodology were suggested by these experiments that might be used to improve the performance of the evolutionary process.

In each study, the evolutionary procedures used only a single problem instance for training. Although not crucial in the first case, the poor translation of training performance to test performance (most notably on the *uuf100* class) suggests that a single instance is not sufficiently representative of the class. Alternatively, the use of a single training instance may too easily allow the evolutionary system to overfit to the training instance. In either case, the use of multiple problem instances would mitigate these effects, and this is one of the modifications made for the empirical study presented in the next chapter.

On the related subject of the assessment of the true performance of the candidate heuristics is the number of repeated runs that should occur in the case of non-deterministic algorithms. Only the first study reported here involved non-deterministic algorithms, and provided for 25 runs per heuristic for evaluation. A subsequent analysis of the results of this experiment revealed that 25 runs was insufficient to truly characterise the performance of a heuristic, and suggests that future work should provide for more extensive evaluation. This is not unrelated to the first point however, as increasing the number of training instances (whilst maintaining a constant number of runs) would similarly serve to eliminate some of the variability that results from the non-determinism of the algorithms. Although the number of runs should be as small as possible in order that the computation cost of training be minimised, a racing algorithm (Birattari et al., 2002) might be used to eliminate algorithms as soon as they could be recognised (with statistical significance) to be inferior, thereby better distributing the available computational resources. Due to technological limitations however, this is not incorporated into the methodology used in the later study.

Finally, these two studies did not use the three stage training-validation-testing process common to both the machine learning and the evolutionary optimisation communities. Although only relevant to the second study, the validation stage was foregone, and the reported test results were simply the best results of the (multiple, one for each training instance) heuristics learned during the training phase. This would tend to overstate the performance of the evolved heuristics, but was unavoidable in order that community-recognised problem sets be used. In many cases these were very small problem sets which did not admit of the generation of large numbers of instances. Practically, this is not a concern, as the computational cost of training is difficult to justify on problem sets with only a few instances. Any practical problem class would therefore provide sufficient instances for independent validation and test sets. This deficiency is addressed in the later empirical study by using independent training, validation and test sets to more correctly assess the performance of the evolved heuristics.

# Chapter 6

# Evolved Algorithms for Hard, Real-World Problems

The previous chapter detailed some initial experiments in evolving algorithms for a variety of constraint problems. The experimental frameworks for those experiments consisted of a basic local search routine, lacking more modern dynamic local search heuristics, and a similarly basic backtracking procedure, lacking all but the simplest forms of consistency maintenace. Although an evolutionary strategy was seen to improve the performance of an initial random population of algorithms, the performance of the evolved algorithm remained inferior to the state-of-the-art. To address this deficiency, the evolutionary adaptive strategy was used in conjunction with a state-of-the-art constraint solver, and this chapter presents results for algorithms evolved within that framework. Furthermore, this study also corrects the items noted at the conclusion of the last chapter, namely the prior use of just single training instances and uses the more rigourous training-validation-testing methodology for evaluating algorithms.

## 6.1   Evolving new heuristics for use with SATZ

SATZ (Li and Anbulagan, 1997) is an algorithm for solving propositional satisfiability problems, based on the Davis-Putnam-Logemann-Loveland (DPLL) procedure (Davis et al., 1962) (Algorithm 4). The DPLL procedure is used for determining the truth or falsity of a boolean propositional formula by recursively instantiating a variable and then simplifying the formula in response to that instantiation. The procedure terminates either when all instantiation paths have been shown to lead to a contradiction (in which case the formula is unsatisfiable), or the formula has no remaining free variables (and is therefore satisfiable).

Primarily, SATZ is an implementation of the DPLL procedure with a specific variable selection heuristic. Perhaps the most novel feature of SATZ when it was originally published

was that it made extensive use of unit-propagation as a branching heuristic, rather than simply as a form of consistency maintenance. Recent SATZ versions also initially perform resolution on the input formula, adding additional clauses into the formula. As an example, if a formula contains the clauses $xyz$ and $\neg xpq$, the resolved clause $pqyz$ may be added to the formula without affecting its satisfiability.

---

**Algorithm 4**: DPLL($F$): DPLL Procedure

**Input**: A set of clauses, $F$. Each clause is a set of literals.
**Output**: $\top$ iff $F$ is satisfiable, otherwise $\bot$.

**while** $F \neq \emptyset$ **do**
  /* Simplify the formula through propagation */
  $F \leftarrow$ FixMonotone($F$);
  $F \leftarrow$ UnitPropagation($F$);

  /* Check for an empty clause (i.e.  a contradiction) */
  **if** $\emptyset \in F$ **then return** $\bot$;

  /* Choose a variable.  If none remain, the formula is satisfiable */
  $x \leftarrow$ VariableSelection($F$);
  **if** $x = \emptyset$ **then return** $\top$;

  /* Attempt both instantiations for the chosen variable */
  **if** DPLL($F \cup \{x\}$) $= \top$ **then**
  |   **return** $\top$;
  **else**
  |   **return** DPLL($F \cup \{\neg x\}$);

---

SATZ operates on a propositional formula by propagating both the positive and negative literals of a variable and recording the reduction these propagations achieve. The SATZ variable selection heuristic is shown as Algorithm 5. As effecting two such propagations for each variable is computationally expensive, SATZ does not always propagate all free variables, but instead propagates only the literals of variables for which the predicate $\mathtt{PROP}_z$ is true.

$\mathtt{PROP}_z$ is defined (at each decision branch) to be the first of the predicates $\mathtt{PROP}_{41}$ or $\mathtt{PROP}_{31}$ that evaluates as true for at least $T$ free variables[1], or finally $\mathtt{PROP}_0$, should neither of the other predicates match sufficient free variables. $\mathtt{PROP}_{41}(x)$ is true, if and only if, $x$ occurs in at least 4 binary clauses, at least once positively and once negatively; similarly, $\mathtt{PROP}_{31}(x)$ is true, if and only if, $x$ occurs in at least 3 binary clauses, at least once positively and once negatively; and finally, $\mathtt{PROP}_0(x)$ is true for any free variable, $x$.

As long as one of these predicates applies (and the inclusion of $\mathtt{PROP}_0$ ensures that one always will), the variable selected to branch on is chosen based on maximising the function

---

[1]In the most recent version of SATZ, satz215.2, $T$ is defined to be 10.

---

**Algorithm 5**: `VariableSelection(F)`: SATZ Variable Selection Heuristic

---

**Input**: A set of clauses, $F$. Each clause is a set of literals.
**Output**: A variable $x \in F$ or $\emptyset$ if $F$ is demonstrated unsatisfiable.
$P \leftarrow \text{PROP}_z(F)$;
**forall** $x \in P$ **do**
    $F' \leftarrow \text{UnitPropagation}(F \cup \{x\})$;
    $F'' \leftarrow \text{UnitPropagation}(F \cup \{\neg x\})$;
    **if** $\emptyset \in F' \wedge \emptyset \in F''$ **then return** $\emptyset$;
    **if** $\emptyset \in F'$ **then** $F \leftarrow F''$;
    **else if** $\emptyset \in F''$ **then** $F \leftarrow F'$;
    **else**
        $w(x) \leftarrow \text{Diff}(F',F)$;
        $w(\neg x) \leftarrow \text{Diff}(F'',F)$;
        $H(x) \leftarrow 1024 * w(x) * w(\neg x) + w(x) + w(\neg x)$;
**return** $x \mid H(x) = \text{max}(H(x))$;

---

$H$, a measure of the amount of reduction that both value assignments would occasion. This is determined for each literal by the function $\text{Diff}(F_1, F_2)$, which returns the number of clauses of minimum size in $F_1$ that are not in $F_2$.

$\text{PROP}_z$ defines a contention function of the three part algorithm representation. Although other variants of such a function could be learned by the evolutionary procedure, these experiments will concentrate exclusively on learning new preference functions to assess the variables that $\text{PROP}_z$ has determined to be in contention.

SATZ's current branching rule is primarily concerned with maximising the reduction in the formula at each branching point. Hooker and Vinay attributed the good performance of such heuristics to the *Simplification Hypothesis*: "Other things being equal, a branching rule works better when it creates simpler subproblems." (Hooker and Vinay, 1995). Whilst in general this may be true, Ouyang has shown instances of pathological structure that quite easily mislead such heuristics (Ouyang, 1996)[2].

This demonstrates though, that irrespective of the lack of applicability of the no free lunch theorems to Davis-Putnam style algorithms, adherence to simplification maximising heuristics does not necessarily lead to the best achievable performance (for all problem structures). This motivates the question of whether the performance of a complex, modern SAT solver might be improved by evolving new branching heuristics for particular classes of problems. This question is addressed in the following study, in which heuristics are evolved for four different classes of increasingly structured benchmark problems.

Given the performance results presented in (Hooker and Vinay, 1995) for literal-based

---

[2]Although in the case of SATZ, the pathological structure described by Ouyang is easily identified as unsatisfiable by its initial process of clause resolution.

heuristics (such as the Jeroslow-Wang rule), there is no reason to maintain the restriction of the original DPLL procedure (and quite peculiarly, SATZ) that the algorithm must always branch to the positive instantiation of a variable first. This allows for the realisation of bicameral heuristics that rank *moves* rather than just variables, and which could foreseeably offer improved performance on some problem classes. For empirical fairness, a modified version of SATZ incorporating the same relaxation is included for comparison (denoted $SATZ_b$).

## 6.2 Methodology

Four different classes of constraint problems are considered in this study: graph 3-colouring problems, random 3-SAT, balanced quasi-groups with holes (BQWH) and the variably modified permutation composition (VMPC) inversion problem. All problems are modelled as propositional satisfiability problems in conjunctive normal form. Details of the specific instance distributions used for each class are discussed in detail in the relevant sub-sections.

The evaluation of the evolutionary system uses a three-part approach, consisting of training, validation and testing stages. Each stage uses independent sets of benchmark problems, in order that heuristics are always evaluated on previously unseen instances.

The available functions and terminals are described in Table 6.1. The evolutionary procedure then proceeds using the following parameters. Initially, 100 instances are selected from a distribution to serve as training instances. For each of the experiments, an initial random population of 100 heuristics is generated. To ensure that sufficient copies of the $+$ and $\times$ functions exist within the population, all individuals in the initial random generation have the form: RandomSubtree1 $\times$ RandomSubtree2 $+$ RandomSubtree3.

Each heuristic in the population is used in place of the current SATZ preference heuristic and evaluated on the training instances. The fitness measure used is the average number of backtracks required to solve the training instances, standardised so that fewer backtracks equates to higher fitness. A (backtrack) cutoff is used to limit the required computational resources. The cutoff is set to the number of backtracks (rounded off) that the original SATZ required to solve 90% of the full set of 100 training instances. Any instances that remain unsolved when the cutoff is reached are assumed to have taken that number of backtracks to solve.

Using the observed fitness values, subsequent populations are generated by standard genetic programming crossover and mutation (Koza, 1990) (producing a further 80 and 10 new individuals respectively) and elitist cloning of the best 10 elements of the population.

| Measures for use in branching heuristics |
|---|
| **Lexico, Shuffle:** Chronological and randomly (but deterministically) reordered chronological branching rules. |
| **NumClauses, ActiveClauses*:** The number of clauses a variable originally occurs in, and those still unsatisfied. |
| **NumBinary*:** As above, but counts active binary clauses. |
| **Degree*, FwdDegree, BwdDegree:** Number of total, uninstantiated and instantiated neighbours of a variable. |
| **PosReduce, NegReduce, MinReduce:** The reduction in the formula from a positive or negative instantiation to this variable, and the minimum of the two. |
| **JeroslowWang, 2SidedJW, RevJW, 1stOrderJW:** The J-W variants described in (Hooker and Vinay, 1995). |
| **WeightedClause*:** Variation of the 2SidedJW heuristic where a clause of length $n-1$ is worth 5 times as much as a clause of length $n$. |
| **UnitClauseCreation*:** The number of clauses that will become unit clauses after enacting this literal. |
| **PosFirst*, NegFirst*:** Measures to enforce a positive- or negative-first literal ordering. PosFirst returns 1 for a positive literal and 0 otherwise (*vice versa* for NegFirst). |
| Plus, Times: Mathematical operators to combine measures. |
| **[-10..-1, 1..10]:** Numerical constants (terminals). |
| **Move:** The variable-value assignment under consideration. |

Table 6.1: Measures for use in branching heuristics. Measures denoted * are either bicameral, or have a bicameral variant. This means they measure properties of a literal rather than of a variable. All measures are scaled to within the range [-1..1], and are applied to the move currently under consideration.

Elements are selected for crossover probabilistically according to their fitness. Those selected for mutation were always the 10 fittest in the population. The termination criteria is that 20 generations have elapsed.

At the conclusion of the training phase, all individuals in the final population progress to the validation phase. Whilst performance on the training instances is a guide to overall performance, the possibility exists that the system has over-fitted the training instances. For this reason, an additional 100 instances are introduced as validation instances. All heuristics are then evaluated using the combined set of training and validation instances. The heuristic exhibiting the best mean backtrack performance on the combined set of training and validation instances is identified as the resultant outcome of the evolutionary experiment for that problem class. Results are presented for the performance of the resultant algorithm compared with the two SATZ heuristics, which includes a statistical comparison using the Wilcoxon Rank-Sum test.

As algorithm run-times often exhibit heavy-tailed behaviour, parametric statistical methods with an underlying assumption of normality are of little use. For this reason, the Wilcoxon Rank-Sum test is used. A one-tailed, paired test is used to test the null hypothesis that the evolved algorithm does not have lower median backtrack performance than SATZ or $SATZ_b$. Small values for the Wilcoxon Rank-Sum probability $p$ allow this hypothesis to

be rejected with a degree of statistical certainty, and we adopt the standard conventions that $p < 0.05$ is significant and $p < 0.01$ is strongly significant.

## 6.3 Experimental Results

### 6.3.1 Evolved Heuristics for SAT-Encoded Colouring Problems

The decision variant of the graph colouring problem involves determining for a graph $G = (V, E)$, if there exists a mapping (colouring) $C : V \rightarrow N$ that assigns to each vertex in $V$ a colour in $N$ such that no two vertices that share an edge in $E$ are assigned the same colour. The relationship between the number of nodes and number of edges in a graph is referred to as connectivity $\gamma$, and satisfies the equation $|E| = 1/2\gamma|V|$. The particular problem distribution selected contains phase transition problems with a connectivity of $\gamma = 4.8$, selected for the difficulty they pose to Brelaz-like smallest-domain heuristics (Hogg, 1996). All instances are guaranteed 3-colourable by originally partitioning the nodes into 3 subsets and explicitly disallowing any edges between nodes in the same partitions (Culberson, 2004). Instances of two sizes are generated. The first set of instances has $|V| = 250$, $|E| = 599$ and $|N| = 3$, and was used for training, validation and testing. A further set of instances with $|V| = 400$, $|E| = 959$ and $|N| = 3$ was used only for testing.

The results of the experiment are presented in Table 6.2 and in Figure 6.1. Backtrack figures represent the average number of backtracks to determine that the problem set is satisfiable. Results for the training stage are for the heuristic that was best on the training set. Similarly, results for the validation stage refer to the heuristic that was best on the combined training and validation sets. The rank column refers to the where the heuristic that performed best of the validation instances ranked on the training instances. Only if the rank column indicates 1 do these figures refer to the same heuristics.

Results for the performance on training instances over time show little apparent improvement after the first few generations. However, the resultant algorithm was evolved as part of the final generation. It cannot therefore be concluded that the experiment may be terminated in this case after just 5 generations.

The evolutionary procedure is capable of learning algorithms that substantially outperform SATZ on the training instances. The best evolved algorithm required only two-thirds the backtracks of SATZ on the training instances. The improvement was less pronounced when evaluation was extended to the validation set, with the best evolved algorithm achieving at best an 8% improvement over the SATZ variants. That this algorithm has not been assigned a rank means that it was generated in the final generation, and evaluated for the
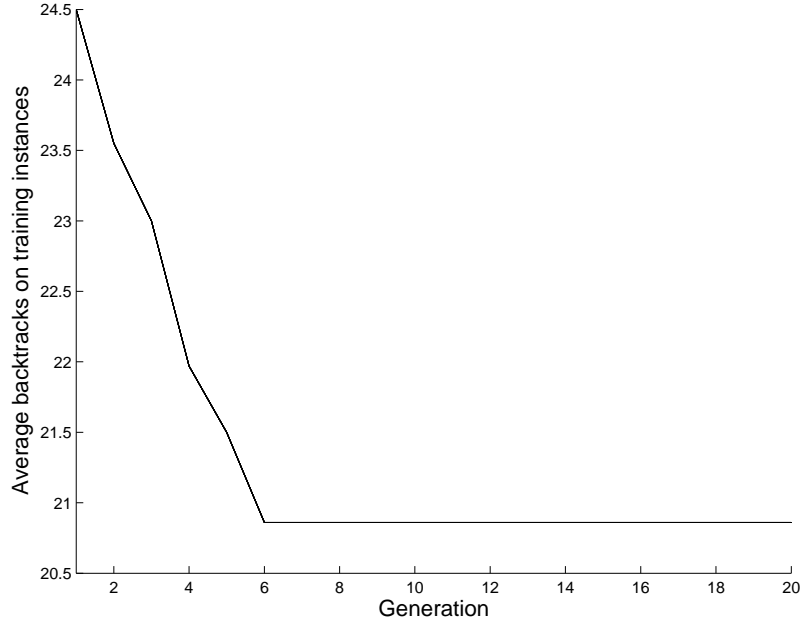
first time on the validation set.



Figure 6.1: Improvement in the performance of the evolved algorithms on graph 3-colouring training instances.

| Training Data & Results | | | Validation Results | | | |
|---|---|---|---|---|---|---|
| SATZ BTs | $\text{SATZ}_b$ BTs | Evolved BTs | Rank of Best | Evolved Avg. BTs | % of SATZ | % of $\text{SATZ}_b$ |
| 20.86 | 30.25 | 30.60 | * | 29.2 | 91.78% | 96.66% |

Table 6.2: Evolved algorithm performance on graph 3-colouring instances.

When evaluated on the 250 node graph instance test set (Table 6.3), the evolved algorithm outperformed SATZ in terms of mean and median backtracks but was outperformed by $\text{SATZ}_b$. On the 400 node test instances, the mean performance of the evolved algorithm was superior to both SATZ variants. The median performance of SATZ was superior to the evolved algorithm however. The Wilcoxon p-value refers to the comparison of the evolved algorithm to each of SATZ and $\text{SATZ}_b$ and according to this the performance differences of these algorithms are not statistically significant. The evolved algorithm is shown in Figure 6.2, but the *move* terminal which is an argument to most of the measures has been omitted for clarity.

## 6.3.2 Evolved Heuristics for Random 3-SAT

Propositional 3-SAT was the original problem to be demonstrated NP-Complete (Cook, 1971). Whilst well-studied and having a known phase transition allowing the generation of
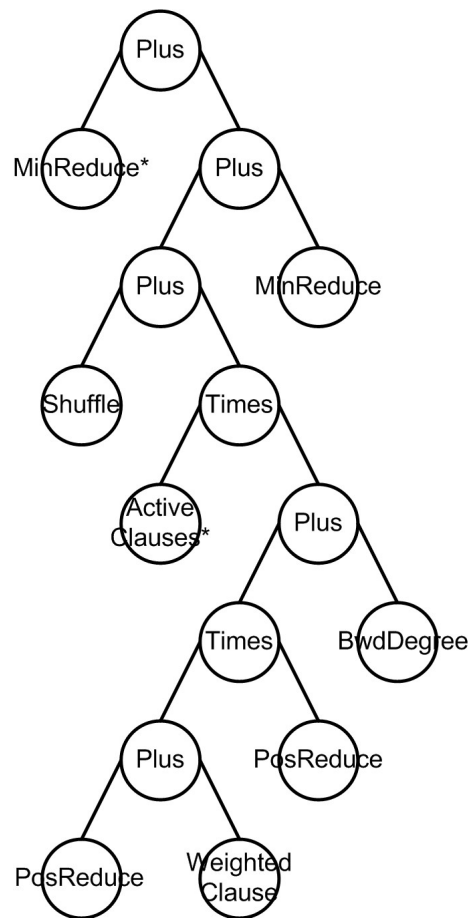
Figure 6.2: The algorithm evolved for graph 3-colouring.

| Class | Algorithm | Mean BTs | Median BTs | Wilcoxon Comparison | |
|---|---|---|---|---|---|
| | | | | p-value | Significant? |
| 250 node | Evolved | 23.94 | 15.00 | | |
| | SATZ | 25.56 | 16.50 | 0.4182 | No |
| | SATZ$_b$ | 23.40 | 12.00 | 0.6241 | No |
| 400 node | Evolved | 1120.82 | 585.00 | | |
| | SATZ | 1150.00 | 476.50 | 0.4230 | No |
| | SATZ$_b$ | 1257.81 | 642.00 | 0.1674 | No |

Table 6.3: Test performance of evolved algorithm for graph 3-colouring.

(on average) difficult instances, it has of late been less popular than other problem types which offer guaranteed-satisfiable instances (like quasigroups with holes, discussed below). Randomly generated 3-SAT instances are not known to be satisfiable in advance, which can create difficulties when using them to test incomplete search procedures, unless they have been filtered by a complete search procedure. Such filtering is often not possible for larger problem instances.

Nonetheless, 3-SAT has remained an important theoretical problem. In this experiment, algorithms tailored for random 3-SAT are evolved. Instances were generated using the same procedure used to generate the uniform filtered (uf) instances available on SATLIB (Hoos and Stützle, 2000). Instances had 250 variables and 1065 clauses, meaning that they were taken from the phase transition region. An additional test set of 300 variable instances with 1270 clauses was also generated. Instances were filtered to ensure satisfiability.

The ability of the evolutionary system to improve performance on the training instances is plotted in Figure 6.3, and the outcome of the training and validation phases are tabulated in Table 6.4. All figures are averages. The results clearly show that evolved algorithms outperform both SATZ and SATZ$_b$ on both the training set and the validation instances. The heuristic that exhibited the best performance on the validation set had been ranked 10th on the training instances.

| Training Data & Results | | | Validation Results | | | |
|---|---|---|---|---|---|---|
| SATZ BTs | SATZ$_b$ BTs | Evolved BTs | Rank of Best | Evolved Avg. BTs | % of SATZ | % of SATZ$_b$ |
| 465.88 | 756.93 | 696.88 | 10 | 577.31 | 79.89% | 79.88% |

Table 6.4: Evolved algorithm performance on random 3-SAT.

The expression for this heuristic is shown in Figure 6.4, as it was evolved and without performing term re-writing or collection. The performance of this algorithm on the test instances compared with SATZ and SATZ$_b$ is tabulated in Table 6.5. Although there is not a
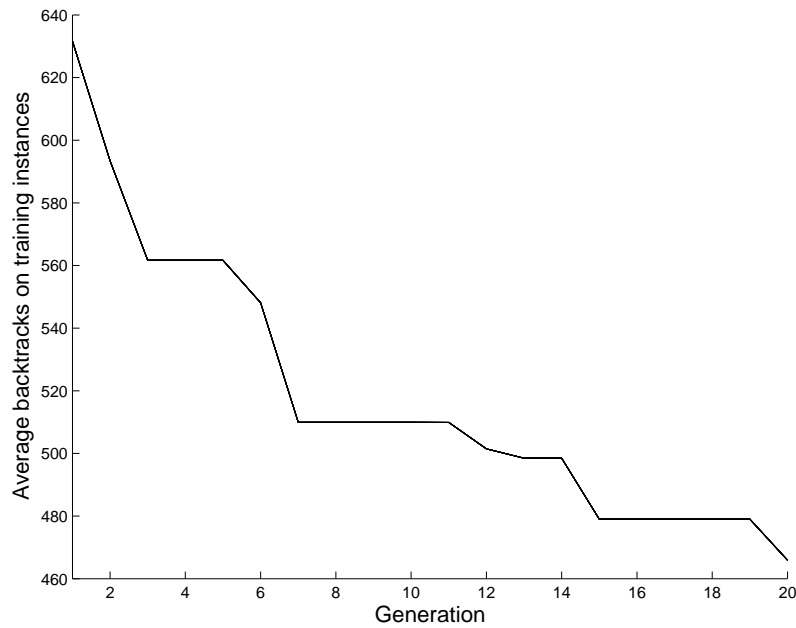
Figure 6.3: Improvement in the performance of the evolved algorithms on random 3-SAT training instances.

great deal of difference in the mean performance of the 3 algorithms, the median performance of the evolved algorithm is substantially better than that of both SATZ and SATZ$_b$. This combination suggests that the distribution characterising the evolved algorithm's run-time performance has a heavier tail than that of the SATZ heuristics. Even though the median performance of the evolved algorithm was at least 25% lower than the alternatives, the Wilcoxon Rank-Sum test did not find the differences in the algorithms' performance to be significantly different.

| Class | Algorithm | Mean BTs | Median BTs | Wilcoxon Comparison | |
|-------|-----------|----------|------------|---------|-------------|
| | | | | p-value | Significant? |
| | Evolved | 612.30 | 307.00 | | |
| 250 var | SATZ | 664.22 | 418.50 | 0.1035 | No |
| | SATZ$_b$ | 617.68 | 425.50 | 0.2325 | No |
| | Evolved | 3629.90 | 2136.00 | | |
| 300 var | SATZ | 4084.63 | 2912.00 | 0.2283 | No |
| | SATZ$_b$ | 4038.33 | 3203.50 | 0.2686 | No |

Table 6.5: Test performance of the evolved algorithm for random 3-SAT.

### 6.3.3 Evolved Heuristics for Balanced Quasigroup Problems

A latin square is an $n$-by-$n$ grid, where the objective is to assign to each cell in the grid a value (or colour) within the range $1..n$. In a valid solution, each colour must appear ex-
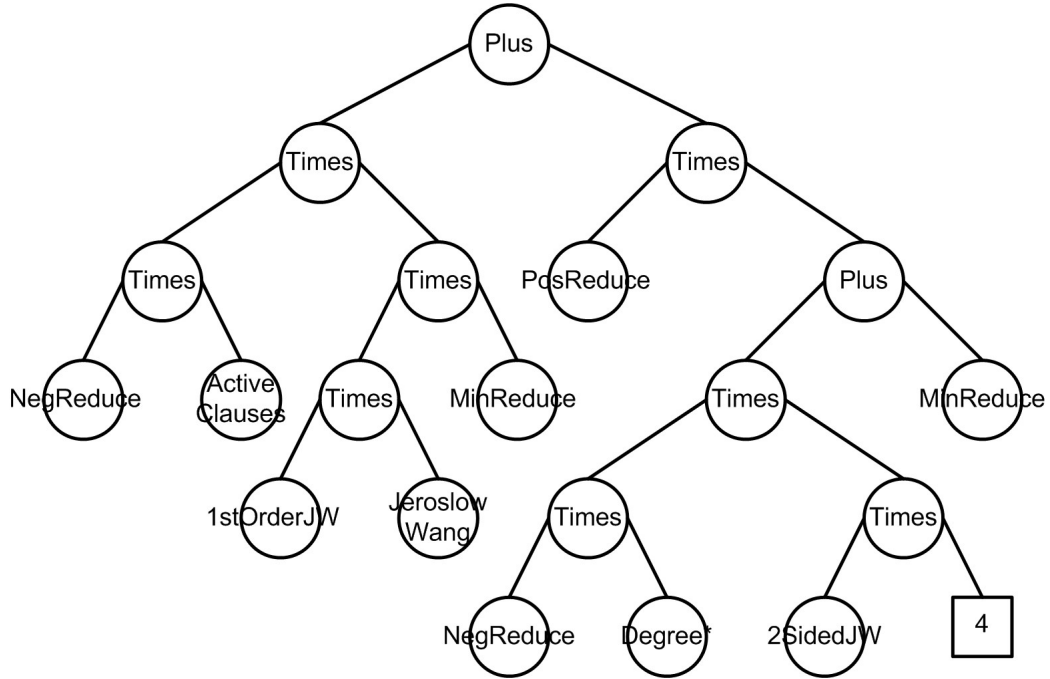
Figure 6.4: The algorithm evolved for random 3-SAT.

actly once in each row and column. A complete, valid latin square defines a solution to the constraint imposed upon a quasigroup: the equations $(a \cdot x) = b$ and $(y \cdot a) = b$ must be uniquely solvable for any pair of elements $a, b$. Quasigroup problems are a highly-structured example of a graph-colouring problem and exhibit a *small-world* topology, meaning that the path length between any two nodes is relatively small (Gomes and Shmoys, 2002). The main consequence of this topology is that the effect of modifying a variable is very rapidly propagated throughout all other parts of the problem. This has traditionally proven particularly problematic for local search procedures, as the rapid propagations make it difficult for such procedures to recognise nearby minima (Anbulagan et al., 2005).

Such structural properties do not invalidate the use of quasigroups as a benchmark problem; instead quite the contrary is true. These problems provide an ideal alternative to 'real-world' problems (which are often in short supply (Achlioptas et al., 2000)) as problem instances can be generated in sufficiently large numbers to permit statistically meaningful studies. Three primary variants of the quasigroup problem have been proposed as benchmark problems: the quasigroup completion problem (QCP) (Gomes and Selman, 1997), which involves determining whether a partially completed solution is extendable to a full solution; and the quasigroup with holes (QWH) (Achlioptas et al., 2000) and the balanced quasigroup

with holes (BQWH) (Kautz et al., 2001) problems, both of which involve completing a so-
lution that is no longer complete due to the introduction of undefined cells termed "holes".
QCP instances are not guaranteed to be satisfiable: consider the trivial example where all
cells but one in a row are specified, entailing the value of the final cell, but this value has also
been specified in a cell of the same column, but a different row. Instances of the other two
classes are guaranteed to be satisfiable, making them particularly suitable as a benchmark
problem for testing incomplete search procedures.

The difference between the QWH and BQWH problems is that balanced instances have
an (approximately) equal number of holes in each row and column, which makes them on
average more difficult than either QCP or QWH instances (Kautz et al., 2001). Furthermore,
the BQWH problem has been well studied and has a known phase transition occuring at
$holes = 1.7 \times order^{1.55}$ (ibid.), making it easy to generate difficult instances. Order 30
problem instances from the known difficult region are used in this study ($holes = 332$,
according to the above equation). These were generated and encoded (using the 3D encoding)
with `lsencode` (Gomes, 2001). Results of evolving algorithms for this problem class are given
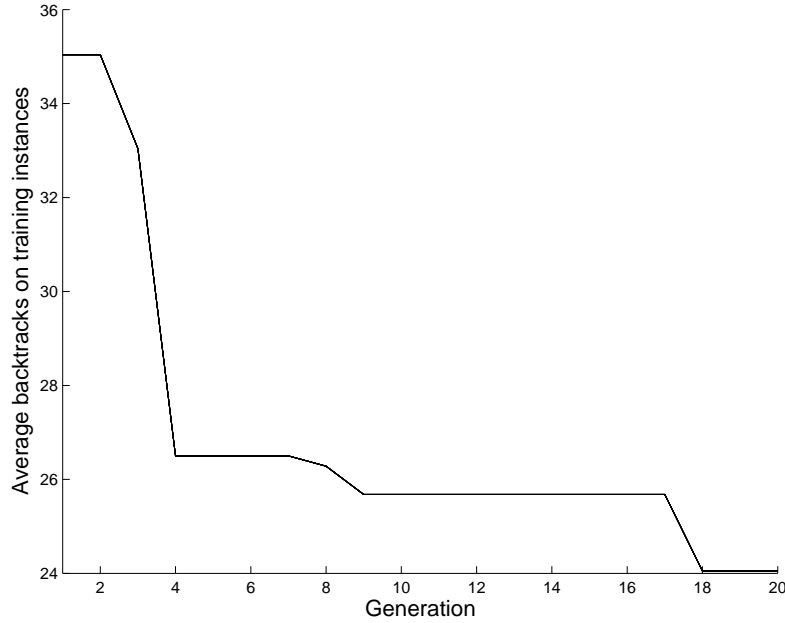in Figure 6.5 and Table 6.6.



Figure 6.5: Improvement in the performance of the evolved algorithms on BQWH training
instances.

The performance of the evolved algorithms on the training instances are considerably
superior to that of either SATZ or SATZ$_b$ and this translates into corresponding good per-
formance on the validation set. The evolved algorithm requiring on average just 53.21%

| Training Data & Results | | | Validation Results | | | |
|---|---|---|---|---|---|---|
| SATZ BTs | $SATZ_b$ BTs | Evolved BTs | Rank of Best | Evolved Avg. BTs | % of SATZ | % of $SATZ_b$ |
| 24.05 | 86.34 | 74.73 | 8 | 39.65 | 53.21% | 49.86% |

Table 6.6: Evolved algorithm performance on balanced quasigroups with holes.

of the backtracks of SATZ and only 49.86% of the backtracks of $SATZ_b$. The specification of this algorithm is given in Figure 6.6. Its performance on the test set is shown in Table 6.7. As has been performed with the two problem classes examined already, the evolved algorithm is also evaluated on larger BQWH instances of orders 30 and 33 as well.

| Class | Algorithm | Mean BTs | Median BTs | Wilcoxon Comparison | |
|---|---|---|---|---|---|
| | | | | p-value | Significant? |
| | Evolved | 37.11 | 7.50 | | |
| Order 27 | SATZ | 113.55 | 23.50 | $3.051 \times 10^{-4}$ | Strongly |
| | $SATZ_b$ | 87.26 | 14.00 | $1.564 \times 10^{-3}$ | Strongly |
| | Evolved | 912.78 | 90.00 | | |
| Order 30 | SATZ | 1987.66 | 288.50 | $1.261 \times 10^{-3}$ | Strongly |
| | $SATZ_b$ | 1718.68 | 329.00 | $3.711 \times 10^{-4}$ | Strongly |
| | Evolved | 28420.25 | 3036.50 | | |
| Order 33 | SATZ | 61634.19 | 8123.00 | $1.519 \times 10^{-4}$ | Strongly |
| | $SATZ_b$ | 60046.53 | 4660.50 | 0.0289 | Yes |

Table 6.7: Test performance of the evolved algorithm for balanced quasigroups with holes.

The evolved algorithm outperforms the SATZ alternatives in terms of both mean and median performance. For both the order 27 and order 30 instances, the Wilcoxon test establishes that the improvement in performance of the evolved algorithm is strongly significant. On the order 33 instances, the improvement over SATZ is strongly significant, but is only significant in comparison with $SATZ_b$. This is attributable to the difference between the median performance of the two SATZ alternatives on this class.

There are a number of possible explanations why significance could be established for the BQWH class but not the the graph 3-colouring and random 3-SAT classes examined earlier. The first is that the SATZ heuristic is simply not suited to this problem. However, comparing SATZ to two other leading complete search algorithms revealed that SATZ (with a mean run-time of 16.44 seconds) was substantially better than either MiniSAT (Eén and Sörensen, 2003) (run-time 27.57 seconds) or zChaff (Moskewicz et al., 2001) (mean run-time of 146.22 seconds).

The BQWH instances are more difficult than those of the other classes considered so far,
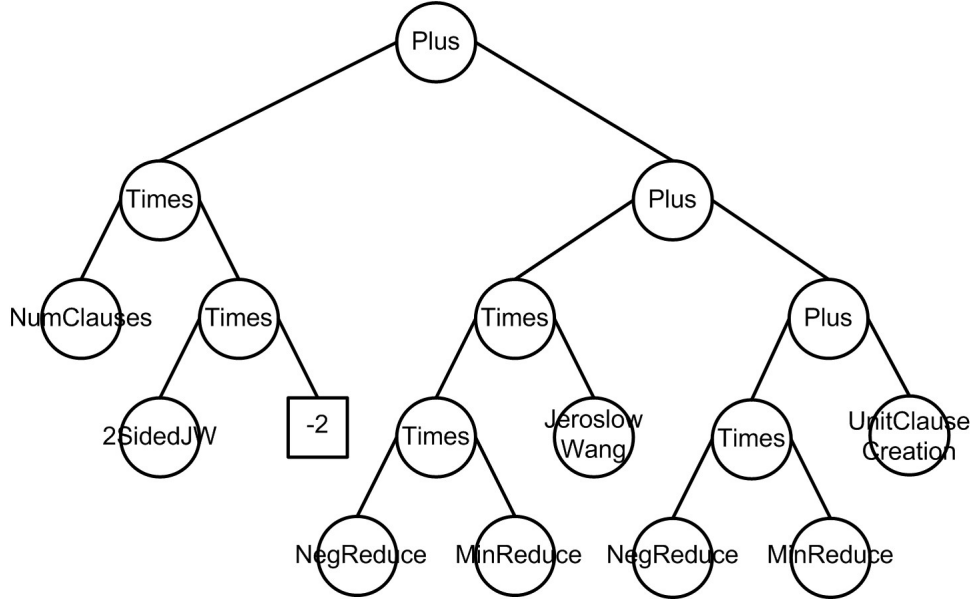
Figure 6.6: Evolved algorithm for balanced quasigroups with holes.

requiring many more backtracks. Perhaps then, the differences are just being more easily identified by the Wilcoxon test. Alternatively, as BQWH is a more structured problem than the other two considered so far, the evolved algorithm may be truly exploiting that structure to achieve better performance.

### 6.3.4   Evolved Heuristics for Cryptographic Problems

The strength of many modern cryptographic methods depends on the difficulty of inverting a variety of one-way functions. Such functions are so-called because they are simple to calculate in one direction only. For example, consider the simple modulus function $r = n\%d$. Given $n$ and $d$ it is trivial to determine $r$. However determining $d$, given $n$ and $r$, requires factorising a product involving $d$. The decision variant of this problem is known to be in $NP \cap co\text{-}NP$, but actually determining what the (prime) factors of $n$ are is believed to be more difficult than the basic decision problem (Garey and Johnson, 1979).

The particular cryptographic problem considered here is that of finding an inversion of a "Variably Modified Permutation Composition" transformation (VMPC) (Zoltak, 2004). Given a permutation $Q$ on a set of non-negative integers $S$, the VMPC inversion problem is to find a permutation $P$ such that $Q[x] = P[(P[P[x]] + 1)\%n]$. Although this is a numerical rather than a boolean problem, it may be quite easily translated to SAT. The resultant SAT
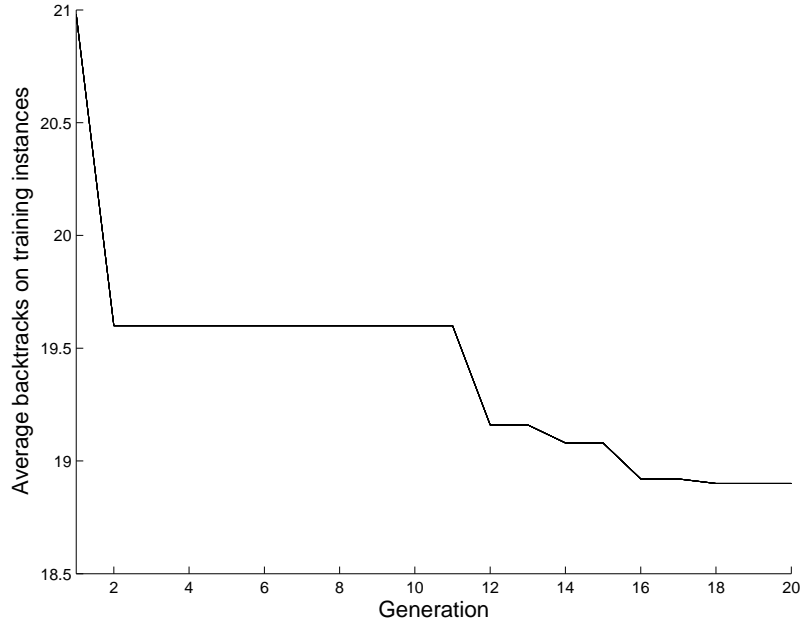
Figure 6.7: Improvement in the performance of the evolved algorithms on VMPC training instances.

instances have $\mathcal{O}(n^2)$ boolean variables and $\mathcal{O}(n^3)$ clauses (Grieu, 2005). 16 such instances were included in the 2005 SAT Competition, with values of $n$ ranging from 21 to 36.

Instances of this problem are substantially more difficult in terms of time (at least for the higher values of $n$) than any of the benchmark instances previously considered. This is due in part simply to the size (number of clauses) in these instances. In the problem classes examined earlier, the use of larger test instances served mainly to demonstrate the generality of the evolved algorithms. For this class the use of smaller instances for training is almost a computational necessity. Problems of size $n = 21$ are used for training and validation, but testing is also conducted on problems of sizes $n = 24, 25, 26$. The progress of the evolution is displayed in Figure 6.7 and results of training and validation are tabulated in Table 6.8. Results are not shown for $SATZ_b$ as it behaves identically to the non-bicameral version on this particular problem's encoding.

| Training Data & Results | | | Validation Results | | | |
|---|---|---|---|---|---|---|
| SATZ BTs | $SATZ_b$ BTs | Evolved BTs | Rank of Best | Evolved Avg. BTs | % of SATZ | % of $SATZ_b$ |
| 29.31 | N/A | 18.90 | * | 29.32 | 77.69% | N/A |

Table 6.8: Evolved algorithm performance on VMPC inversion problems.

Test results for the evolved algorithm for VMPC inversion are presented in Table 6.9. The mean backtracks required by the evolved algorithm was in all cases fewer than the number

required by SATZ. The median backtracks required was fewer in all cases except for the $n = 26$ instances. Furthermore, the superiority of the evolved algorithm can be established with strong statistical significance in two of the four cases. That one of these cases is the simplest instance of this type would tend to refute the possibility expressed in the previous section that demonstrating statistical significance requires 'hard' problem instances.

| Class | Algorithm | Mean BTs | Median BTs | Wilcoxon Comparison | |
|---|---|---|---|---|---|
| | | | | p-value | Significant? |
| Size 21 | Evolved | 28.21 | 6.50 | $1.214 \times 10^{-4}$ | Strongly |
| | SATZ | 32.00 | 8.50 | | |
| Size 24 | Evolved | 246.34 | 101.00 | 0.1322 | No |
| | SATZ | 252.20 | 103.50 | | |
| Size 25 | Evolved | 391.48 | 212.00 | $1.079 \times 10^{-2}$ | Strongly |
| | SATZ | 479.32 | 230.50 | | |
| Size 26 | Evolved | 709.47 | 356.00 | 0.1301 | No |
| | SATZ | 837.99 | 330.00 | | |

Table 6.9: Test performance of the evolved algorithm for VMPC inversion.
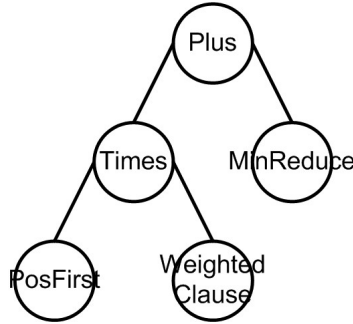


Figure 6.8: Evolved algorithm for the VMPC inversion problem.

## 6.4 Discussion

These experiments have examined four different classes of constraint problem posed as propositional satisfiability problems. The evolutionary procedure was repeatedly shown to evolve heuristics that outperformed the baseline SATZ heuristics on the training instances. When extended to validation and test sets, often involving larger, more difficult problem instances, the evolved algorithms were generally shown to also exhibit superior performance.

Although the Wilcoxon Rank-Sum test was unable to establish the superiority of the evolved algorithms with statistical certainty in either the random 3-SAT or graph 3-colouring

classes, these problem classes are not inherently structured and furthermore, are not overly difficult. In both the BQWH and the VMPC problem classes though, the Wilcoxon Test provided strong statistical evidence for the superiority of the evolved algorithms.

It remains a matter for future work to examine why there is less distinction between the performance of the evolved algorithms and existing methods on some problem classes. The efficacy of an evolutionary approach has however been demonstrated on exactly the types of problems that are considered representative of hard, real-world instances.

# Chapter 7

# Conclusions and Future Work

## 7.1 Conclusions

One of the great challenges facing the constraints community is to develop methods that more easily allow constraint programming technologies to be integrated into the applications of end-users. The goal of this work has been to address this challenge by developing a method for the automatic development of heuristics for particular classes of constraint problems. Such a system has been realised and demonstrated to achieve statistically significant superior performance over existing state-of-the-art methods. A number of specific contributions have been made in order to realise this goal.

Firstly, a new representation for constraint algorithms was proposed. This representation is sufficiently expressive to represent the wide range of algorithms that are currently in use. Furthermore, it is not limited to a particular search paradigm, being suitable for both complete and local search procedures.

Secondly, genetic programming, an evolutionary method of adaptation, was suggested as an ideal method for use with this representation. This is believed to be the first use of pure genetic programming for the automatic adaptation of constraint algorithms.

The proposed representation and the use of genetic programming were each motivated by the recognition of five important features of an adaptive system. These features - versatility, creativity, expressivity, foresight and hindsight - have been present in isolation in a number of existing works. The system demonstrated in this work exhibits all five features. This has been realised by combining genetic programming with the proposed respresentation.

Finally, a systematic and rigorous statistical evaluation of the proposed system has shown that it can automatically develop improved algorithms for a wide variety of constraint problems. Using the same method, heuristics for other types of constraint problems may now be similarly realised - enitrely without the involvement of human constraint solving expertise.

## 7.2   Future Work

A number of ideas for potential future work are discussed in the following sections.

### 7.2.1   Improving the efficiency of the evolutionary procedure

Birattari's racing algorithm (Birattari et al., 2002) is not a new method of adaptation, but a method to identify statistically significant below average performance as soon as possible in order to more efficiently allocate computational resources.

In the context of a genetic system, such a method could potentially diminish the genetic diversity of the population if under-performing algorithms were immediately removed (intra-generation) from the population, potentially eliminating the ability of such a system to recognise synergies. The most obvious solution to this is to retain the under-performing algorithms in the population during the current generation, but assign them no additional computational resources. Provided that the fitness measure relied upon the same indicators used to identify heuristics as under-performers, different allocations of computational resources to candidate heuristics should not matter.

Alternatively, Birrataris procedure would be ideally suited for use a rank fitness measure, as the fitness value assigned to each heuristic could simply reflect the order in which it was eliminated from further consideration. This has the advantage over the above method in that it does not rely on an explicit determination of fitness. That is, although an algorithm may have been demonstrated statistically inferior, this does not mean that its true fitness has been predicted in absolute terms. Only its relative fitness with respect to the other algorithms in the population has been determined. As rank fitness measures are entirely relative, this limitation does not affect them.

That such procedures were not used in this work was primarily due to the technological constraints of the available computing resources, but would certainly warrant investigation for use in a production, rather than an experimental, system.

### 7.2.2   Alternative fitness measures

The fitness measures used in these studies have been a function of the run-time or run-length performance of the candidate heuristics. Being an objective measure of the performance of an algorithm, it is the predominant metric used in the literature to compare algorithms, and is therefore a natural choice for the objective function of an adaptive constraint system. After all, the cardinal rule of evolutionary computing (and indeed all optimisation) is that a system is only as good as its fitness function, and the only elements that such a system

will optimise will be those reflected in its fitness function. Since run-time is arguably the most important measure to the constraint satisfaction community, it is the ideal choice as a fitness measure. But is this the only fitness measure that might be used?

There are bodies of literature in both the local and complete search communities that attempt to explain the improved performance of some algorithms over others. Schuurmans et al. proposed three metrics (depth, mobility and coverage) to characterise the behaviour of local search routines (Schuurmans et al., 2001). Particular combinations of these were seen to be indicative of efficient search procedures. Similarly, a number of explanations have been presented for the behaviour of complete search routines. Jeroslow and Wang suggested that it was important for the search to branch into satisfiable sub-problems (Jeroslow and Wang, 1990). This was later discredited by Hooker and Vinay, who suggested that the simplification of the resultant sub-problems was more important (Hooker and Vinay, 1995). Although both of these works were exclusively concerned with the propositional satisfiability problem, there have been similar pronouncements for the more general CSP. The concept of fail-firstness, suggested by (Haralick and Elliott, 1980) and explored in some depth by Smith and Grant (Smith and Grant, 1998), is that effective heuristics should "try first where they expect to fail. In some respects, it is a combination of the opposite of Jeroslow and Wangs satisfaction hypothesis (that is, search should branch into unsatisfiable sub-problems) and Hooker and Vinays simplification hypothesis (the earlier that failure occurs, the better).

The essential feature of these metrics (with the exception of Jeroslow and Wangs discredited satisfaction hypothesis) is that they do not require the overall satisfaction of a test instance to be determined in order to characterise the performance of a heuristic. They can therefore be used to assess performance during search, rather than either waiting for the search to complete or aborting the search for lack of progress. Given the computational cost of adaptive procedures, any methods that might improve the efficiency of such procedures are deserving of further study.

Furthermore, a study investigating methods to better estimate the overall run-time of an algorithm from partial performance will have important implications for the online adaptive search community.

### 7.2.3 Learning from learning

Whilst evolutionary and adaptive systems may produce improved algorithms, if characterised solely by their run-time performance, then to constraint practitioners they exist as little more than black-boxes: they might perform well, but why? After all, improving the run-time performance of an algorithm is only the destination, and need not necessarily be part

of the journey.

It is then perhaps worthwhile to ask what learned algorithms might tell us about particular problem classes in specific, but moreover about constraint satisfaction problems in general.

Given the facility to automatically realise large populations of algorithms through evolution or any other method; a larger sample than might ever result from manual human endeavour; might it not also be possible to analyse the behaviour of these algorithms to characterise not only how well they perform, but why?

This is certainly an ambitious goal, and was not touched upon in the current work; it presents what is most likely an extraordinarily difficult unsupervised machine-learning problem. It necessarily involves learning functions that characterise the behaviour of algorithms, rather than the algorithms themselves.

As outwardly ambitious as such a proposal might seem, it is perhaps reassuring to draw an analogy: learning the meta-level processes responsible for the underlying performance of algorithms can be likened to learning the meta-level algorithms responsible for determining the underlying solutions to a constraint problem.

Since achievement of the latter has hopefully been demonstrated by this thesis, it is not unreasonable to consider the former an achievable goal for the constraint satisfaction community.

# References

[Achlioptas et al., 2000] Dimitris Achlioptas, Carla P. Gomes, Henry A. Kautz, and Bart Selman. Generating satisfiable problem instances. In *AAAI'00*, pages 256–261, 2000.

[Anbulagan et al., 2005] Anbulagan, Duc Nghia Pham, John Slaney, and Abdul Sattar. Old resolution meets modern SLS. In *AAAI'05*, pages 354–359, 2005.

[Bäck et al., 1991] T. Bäck, F. Hoffmeister, and H. Schwefel. A survey of evolution strategies. In *Proceedings of the 4th International Conference on Genetic Algorithms*, pages 2–9, 1991.

[Bain et al., 2004a] Stuart Bain, John Thornton, and Abdul Sattar. Evolving algorithms for constraint satisfaction. In *Proc. of the 2004 Congress on Evolutionary Computation*, pages 265–272, 2004a.

[Bain et al., 2004b] Stuart Bain, John Thornton, and Abdul Sattar. Methods of automatic algorithm generation. In *PRICAI'04*, pages 144–153, 2004b.

[Bain et al., 2005] Stuart Bain, John Thornton, and Abdul Sattar. Evolving variable-ordering heuristics for constrained optimisation. In *CP'05*, pages 732–736, 2005.

[Baker, 1987] James E. Baker. Reducing bias and inefficiency in the selection algorithm. In *Proceedings of the 2nd International Conference on Genetic Algorithms*, pages 14–21. Lawrence Erlbaum Associates, 1987.

[Beringer et al., 1994] A. Beringer, G. Aschemann, H. Hoos, M. Metzger, and A. Weiss. GSAT versus simulated annealing. In *ECAI'94*, 1994.

[Bessière and Régin, 1996] Christian Bessière and Jean-Charles Régin. MAC and combined heuristics: Two reasons to forsake FC (and CBJ?) on hard problems. In *CP'96*, pages 61–75, 1996.

[Birattari et al., 2002] Mauro Birattari, Thomas Stützle, Luis Paquete, and Klaus Varrentrapp. A racing algorithm for configuring meta-heuristics. In *GECCO'02*, pages 11–18, 2002.

[Bistarelli et al., 1995] Stefano Bistarelli, Ugo Montanari, and Francesca Rossi. Constraint solving over Semirings. In *IJCAI '95*, pages 624–630, 1995.

[Bistarelli et al., 1997] Stefano Bistarelli, Ugo Montanari, and Francesca Rossi. Semiring-based constraint satisfaction and optimization. *JACM*, 44(2):201–236, 1997.

[Bobrow and Raphael, 1974] D. G. Bobrow and B. Raphael. New programming languages for AI research. *Computer Survey*, 6:153–174, 1974.

[Boden, 1994] Margaret Boden. Creativity and computers. In Terry Dartnall, editor, *Artificial Intelligence and Creativity*. Kluwer Academic Publishers, 1994.

[Bohlin, 2002] Markus Bohlin. Constraint programming graduate course: Heuristic methods, 2002.
URL: http://www.idt.mdh.se/phd/courses/constraints/slides/locsearch.pdf.

[Borchers and Furman, 1999] B. Borchers and J. Furman. A two-phase exact algorithm for MAX-SAT and weighted MAX-SAT problems. *Journal of Combinatorial Optimization*, 2: 299–306, 1999.

[Borrett et al., 1996] J. E. Borrett, Edward P. K. Tsang, and N. R. Walsh. Adaptive constraint satisfaction: The quickest first principle. In *European Conference on Artificial Intelligence*, pages 160–164, 1996.

[Braunstein and Zecchina, 2004] Alfredo Braunstein and Riccardo Zecchina. Survey and belief propagation on random k-SAT. *Lecture Notes in Computer Science*, 2919:519–528, 2004.

[Caseau et al., 1999] Yves Caseau, François Laburthe, and Glenn Silverstein. A meta-heuristic factory for vehicle routing problems. In *CP'99*, pages 144–159, 1999.

[Cha and Iwama, 1997] Byungki Cha and Kazuo Iwama. Adding new clauses for faster local search. In *AAAI'97, Vol. 1*, pages 332–337, 1997.

[Cook, 1971] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proc. of the Third Annual ACM Symposium on Theory of Computing*, pages 151–158, 1971.

[Culberson, 2004] Joseph Culberson. Culberson's hidden k-colorable graph generator, June 2004.
URL: http://web.cs.ualberta.ca/ joe/Coloring/Generators/generate.html.

[Davis et al., 1962] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 5(7):394–397, 1962.

[Davis and Putnam, 1960] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, 1960.

[Dechter and Frost, 1998] R. Dechter and D. Frost. Backtracking algorithms for constraint satisfaction problems - a survey. Technical report, UCI, http://www.ics.uci.edu/ dechter/, 1998.

[Dechter, 2003] Rina Dechter. *Constraint Processing*. Morgan Kaufmann, 2003. ISBN 1-55860-890-7.

[Eckhardt, 1987] R. Eckhardt. Stan Ulam, John von Neumann, and the Monte Carlo Method. *Los Alamos Science*, Special Issue dedicated to Stanislaw Ulam:125–130, 1987.

[Eén and Sörensen, 2003] Niklas Eén and Niklas Sörensen. An extensible SAT solver. In *SAT 2003*, pages 502–518, 2003.

[Epstein et al., 2002] Susan L. Epstein, Eugene C. Freuder, Richard Wallace, Anton Morozov, and Bruce Samuels. The adaptive constraint engine. In *CP'02*, pages 525–540, 2002.

[Frank, 1996] Jeremy Frank. Weighting for Godot: Learning heuristics for GSAT. In *AAAI'97*, volume 1, pages 338–343, 1996.

[Frank, 1997] Jeremy Frank. Learning short-term weights for GSAT. In *IJCAI '97*, pages 384–391, 1997.

[Freeman, 1995] J.W. Freeman. *Improvements to propositional satisfiability search algorithms*. PhD thesis, Dept. of Computer Science, University of Pennsylvania, 1995.

[Freuder and Wallace, 1992] Eugene C. Freuder and Richard J. Wallace. Partial constraint satisfaction. *Artificial Intelligence*, 58:21–70, 1992.

[Frisch and Peugniez, 2001] Alan M. Frisch and Timothy J. Peugniez. Solving non-boolean satisfiability problems with stochastic local search. In *IJCAI'01*, pages 282–290, 2001.

[Frost and Dechter, 1994] Daniel Frost and Rina Dechter. In search of the best constraint satisfaction search. In *AAAI'94*, pages 301–306, 1994.

[Fukunaga, 2002] Alex Fukunaga. Automated discovery of composite SAT variable-selection heuristics. In *Proceedings of AAAI 2002*, pages 641–648, 2002.

[Fukunaga, 2004] Alex Fukunaga. Evolving local search heuristics for SAT. In *GECCO-04*, 2004.

[Garey and Johnson, 1979] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, San Francisco, 1979.

[Gaschnig, 1978] John Gaschnig. Experimental case studies of backtrack vs. waltz-type vs. new algorithms for satisficing assignment problems. In *Proceedings of the Second Canadian Conference on Artificial Intelligence*, pages 268–277, 1978.

[Génisson and Jégou, 1996] Richard Génisson and Philippe Jégou. Davis and Putnam were already checking forward. In *ECAI'96*, pages 180–184, 1996.

[Gent, 2002] Ian P. Gent. Arc consistency in SAT. Technical Report APES-39-2002, Algorithms, Problems, and Empirical Studies, 2002.
URL: `http://www.dcs.st-and.ac.uk/∼apes/ reports/apes-39-2002.pdf`.

[Ginsberg and McAllester, 1994] M. L. Ginsberg and D. A. McAllester. GSAT and Dynamic Backtracking. In *KR 1994*, pages 226–237. Morgan Kaufmann, 1994.

[Ginsberg, 1999] Matthew L. Ginsberg. Systematic and non-systematic search. In *IJCAI'95, Panel Discussion: Systematic versus Stochastic Constraint Satisfaction*, volume 2, pages 2027–2032, 1999.

[Glover, 1989] Fred Glover. Tabu search - Part I. *ORSA Journal of Computing*, 1(3):190–206, 1989.

[Glover, 1990] Fred Glover. Tabu search - Part II. *ORSA Journal of Computing*, 2(1):4–32, 1990.

[Goldberg, 1989] David Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley, 1989.

[Goldberg et al., 1989] David E. Goldberg, Bradley Korb, and Kalyanmoy Deb. Messy genetic algorithms: Motivation, analysis, and first results. *Complex Systems*, 3(5):493–530, 1989.

[Golomb and Baumert, 1965] Solomon W. Golomb and Leonard D. Baumert. Backtrack programming. *Journal of the ACM*, 12(4):516–524, 1965.

[Gomes and Selman, 2001] Carla Gomes and Bart Selman. Algorithm portfolios. *Artificial Intelligence*, 126:43–62, 2001.

[Gomes, 2001] Carla P. Gomes. Generator of quasigroup completion problem and related problems, *http://www.cs.cornell.edu/gomes/soft/lsencode-v1.1.tar.z*, 2001. URL: `http://www.cs.cornell.edu/gomes/SOFT/lsencode-v1.1.tar.Z`.

[Gomes and Selman, 1997] Carla P. Gomes and Bart Selman. Problem structure in the presence of perturbations. In *AAAI'97*, pages 221–226, 1997.

[Gomes and Shmoys, 2002] Carla P. Gomes and David Shmoys. Completing quasigroups or latin squares: A structured graph coloring problem. In *Proc. Computational Symposium on Graph Colouring and Generalizations*, pages 22–39, 2002.

[Gomes et al., 1997] Carla P. Gomes, Bart Selman, and Nuno Crato. Heavy-tailed distributions in combinatorial search. In *CP'97*, pages 121–135, 1997.

[Gratch and Chien, 1996] Jonathan Gratch and Steve Chien. Adaptive problem-solving for large-scale scheduling problems: A case study. *Journal of Artificial Intelligence Research*, 4:365–396, 1996.

[Gratch and DeJong, 1992] Jonathon Gratch and Gerald DeJong. COMPOSER: A probabilistic solution to the utility problem in speed up learning. In *AAAI '92*, pages 235–240, 1992.

[Gratch and DeJong, 1996] Jonathon Gratch and Gerald DeJong. A statistical approach to adaptive problem solving. *Artificial Intelligence*, 88:101–142, 1996.

[Grieu, 2005] François Grieu. VMPC inversion problems, 2005.

[Hajek, 1988] Bruce Hajek. Cooling schedules for optimal annealing. *Mathematics of Operations Research*, 13(2):311–329, 1988.

[Han and Kendall, 2003] Limin Han and Graham Kendall. An investigation of a Tabu assisted hyper-heuristic genetic algorithm. In *2003 Congress on Evolutionary Computation*, volume 3, pages 2230–2237. IEEE Press, 2003.

[Hansen and Jaumard, 1990] J. Hansen and B. Jaumard. Algorithms for the maximum satisfiability problem. *Computing*, 44:279–303, 1990.

[Haralick and Elliott, 1980] R. Haralick and G. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980.

[Hogg, 1996] Tad Hogg. Refining the phase transition in combinatorial search. *Artificial Intelligence*, 81:127–154, 1996.

[Holland, 1975] John H. Holland. *Adaptation in natural and artificial systems*. University of Michigan Press, 1975.

[Holland, 1992] John H. Holland. *Adaptation in natural and artificial systems, 2nd Edition*. MIT Press, Cambridge, Massachusetts, 1992.

[Hooker and Vinay, 1995] John N. Hooker and V. Vinay. Branching rules for satisfiability. *Journal of Automated Reasoning*, 15(3):359–383, 1995.

[Hoos, 2002] Holger H. Hoos. An adaptive noise mechanism for WalkSAT. In *AAAI'02*, pages 655–660, 2002.

[Hoos, 1998] Holger H. Hoos. *Stochastic Local Search - Methods, Models, Applications*. PhD thesis, Computer Science Department of the Darmstadt University of Technology, 1998.

[Hoos and Stützle, 2000] Holger H. Hoos and Thomas Stützle. SATLIB: An online resource for research on sat. In *SAT 2000*, pages 283–292, 2000.
URL: http://www.satlib.org.

[Hoos and Stützle, 2005] Holger H. Hoos and Thomas Stützle. *Stochastic Local Search: Foundations and Applications*. Morgan Kaufmann, 2005.

[Hutter et al., 2002] F. Hutter, D. Tompkins, and H. Hoos. Scaling and probabilistic smoothing: Efficient dynamic local search for SAT. In *CP'02*, pages 233–248. Springer Verlag, 2002.

[Hutter and Hamadi, 2005] Frank Hutter and Youssef Hamadi. Parameter adjustment based on performance prediction: Towards an instance-aware problem solver. Technical Report MSR-TR-2005-125, Microsoft Research, December 2005.

[Jampel et al., 1996] Michael Jampel, Eugene Freuder, and Michael Maher, editors. *Over-Constrained Systems*, volume 1106 of *LNCS*. Springer-Verlag, 1996.

[Jeroslow and Wang, 1990] R. G. Jeroslow and J. Wang. Solving propositional satisfiability problems. *Annals of Mathematics and Artificial Intelligence*, 1:167–187, 1990.

[Kasif, 1990] Simon Kasif. On the parallel complexity of discrete relaxation in constraint satisfaction networks. *Artificial Intelligence*, 45(3):275–286, 1990.

[Kautz et al., 2001] Henry A. Kautz, Yongshao Ruan, Dimitris Achlioptas, Carla P. Gomes, Bart Selman, and Mark E. Stickel. Balance and filtering in structured satisfiable problems. In *IJCAI'01*, pages 351–358, 2001.

[Kirkpatrick et al., 1983] S. Kirkpatrick, C. D. Jr. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[Koza, 1990] John R. Koza. Genetic Programming: A paradigm for genetically breeding populations of computer programs to solve problems. Technical Report STAN-CS-90-1314, Stanford University Computer Science Department, 1990.

[Koza, 1992] John R. Koza. *Genetic Programming: On the programming of computers by means of natural selection*. MIT Press, Cambridge, Massachusetts, 1992.

[Koza, 1994] John R. Koza. *Genetic Programming II*. MIT Press, Cambridge, Massachusetts, 1994.

[Laburthe and Caseau, 1998] François Laburthe and Yves Caseau. SALSA: A language for search algorithms. *Lecture Notes in Computer Science*, 1520:310–324, 1998.

[Le Berre and Simon, 2005a] Daniel Le Berre and Laurent Simon. The SAT 2005 competition: Fourth edition. In *SAT'05*, 2005a.

[Le Berre and Simon, 2005b] Daniel Le Berre and Laurent Simon. The SAT 2005 Competition. Presented at SAT'05, 2005b.
URL: http://www.satcompetition.org/2005/presentation-last.pdf.

[Leyton-Brown et al., 2005] Kevin Leyton-Brown, Eugene Nudelman, and Yoav Shoham. Empirical hardness models: Methodology and a case study on combinatorial auctions. *Under review by the Journal of the ACM*, 2005.

[Li and Anbulagan, 1997] Chu Min Li and Anbulagan. Heuristics based on unit propagation for satisfiability problems. In *IJCAI'97*, pages 366–371, 1997.

[Mackworth, 1977] Alan Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.

[McAllester et al., 1997] David McAllester, Bart Selman, and Henry Kautz. Evidence for invariants in local search. In *AAAI'97*, pages 321–326, 1997.

[Mills and Tsang, 2000] P. Mills and E. Tsang. Guided local search for solving SAT and weighted MAX-SAT problems. *Journal of Automated Reasoning*, 24:205–223, 2000.

[Minton, 1993] Steven Minton. An analytic learning system for specializing heuristics. In *IJCAI '93*, pages 922–929, 1993.

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

[Minton et al., 1992a] Steven Minton, Mark D. Johnston, Andrew B. Philips, and Philip Laird. Solving large-scale constraint satisfaction scheduling problems using a heuristic repair method. In *AAAI'90*, San Jose, CA, 1992a.

[Minton et al., 1992b] Steven Minton, Mark D. Johnston, Andrew B. Philips, and Philip Laird. Minimizing conflicts: A heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence*, 58:161–205, 1992b.

[Montanari, 1974] Ugo Montanari. Networks of constraints: Fundamental properties and applications to picutre processing. *Information Science*, 7:95–132, 1974.

[Morris, 1993] Paul Morris. The breakout method for escaping from local minima. In *AAAI'93*, page 40ff, 1993.

[Moscato, 1989] Pablo Moscato. On evolution, search, optimization, genetic algorithms and martial arts: Towards memetic algorithms. Technical Report C3P Report 826, 1989.

[Moskewicz et al., 2001] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, 2001.
URL: `citeseer.ist.psu.edu/moskewicz01chaff.html`.

[Nareyek, 2001] Alexander Nareyek. Choosing search heuristics by non-stationary reinforcement learning. In *Metaheuristics: Computer Decision Making*, pages 523–544. Kluwer Academic, 2001.

[Nudelman et al., 2004] Eugene Nudelman, Kevin Leyton-Brown, Holger H. Hoos, Alex Devkar, and Yoav Shoham. Understanding random SAT: Beyond the clauses-to-variables ratio. In *CP'04*, pages 438–452, 2004.

[Olsson, 1995] Roland Olsson. Inductive functional programming using incremental program transformation. *Artificial Intelligence*, 74(1):55–81, 1995.

[Ouyang, 1996] Ming Ouyang. How good are branching rules in DPLL? Technical Report 96-38, DIMACS, 9, 1996.

[Papadimitriou, 1991] C.H. Papadimitriou. On selecting a satisfying truth assignment. In *Conference on the Foundations of Computer Science*, pages 163–169, 1991.

[Pearl, 1984] J. Pearl. *Heuristics: Intelligent search strategies for computer problem solving.* Addison Wesley, 1984.

[Poli, 2001a] Riccardo Poli. Exact schema theory for genetic programming and variable-length genetic algorithms with one-point crossover. *Genetic Programming and Evolvable Machines*, 2(2):123–163, 2001a.

[Poli, 2001b] Riccardo Poli. General schema theory for genetic programming with subtree-swapping crossover. In *LNCS 2038: EuroGP 2001*, pages 143–159, 2001b.

[Poli and Langdon, 1998] Riccardo Poli and William B. Langdon. Schema theory for genetic programming with one-point crossover and point mutation. *Evolutionary Computation Journal*, 6(3):231–252, 1998.

[Puget, 2004] Jean-François Puget. Constraint Programming's next challenge: Simplicity of use. *Invited talk.* In *CP'04*, 2004.

[Purdom, 1983] Paul Purdom. Search rearrangement backtracking and polynomial average time. *Artificial Intelligence*, 21:117–133, 1983.

[Rice, 1976] John R. Rice. The algorithm selection problem. *Advances in Computers*, 15: 65–118, 1976.

[Sabin and Freuder, 1994] Daniel Sabin and Eugene C. Freuder. Contradicting conventional wisdom in constraint satisfaction. In *ECAI'94*, pages 125–129, 1994.

[Schuurmans and Southey, 2001] Dale Schuurmans and Finnegan Southey. Local search characteristics of incomplete SAT procedures. *Artificial Intelligence*, 132(2):121–150, 2001.

[Schuurmans et al., 2001] Dale Schuurmans, Finnegan Southey, and Robert C. Holte. The exponentiated subgradient algorithm for heuristic boolean programming. In *IJCAI '01*, pages 334–341, 2001.

[Selman and Kautz, 1993] Bart Selman and Henry A. Kautz. Domain-independent extensions to GSAT: Solving large structured satisfiability problems. In *IJCAI-93*, 1993.

[Selman et al., 1992] Bart Selman, Hector J. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. In *AAAI'92*, pages 440–446. AAAI Press, 1992.

[Selman et al., 1994] Bart Selman, Henry A. Kautz, and Bram Cohen. Noise strategies for improving local search. In *AAAI'94*, pages 337–343, 1994.

[Shang and Wah, 1998] Y. Shang and B. Wah. A discrete lagrangian-based global-search method for solving satisfiability problems. *Journal of Global Optimization*, 12(1):61–100, 1998.

[Smith and Grant, 1998] Barbara M. Smith and Stuart A. Grant. Trying harder to fail first. In *ECAI'98*, pages 249–253, 1998.

[Spears, 1996] William Spears. Simulated annealing for hard satisfiability problems. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 533–558, 1996.

[Thornton et al., 2004] John Thornton, D. Nghia Pham, Stuart Bain, and Valnir Ferreira. Additive versus multiplicative clause weighting for SAT. In *AAAI'04*, pages 191–196, 2004.

[Turing, 1950] Alan Turing. Computing machinery and intelligence. *Mind*, 59:433–460, 1950.

[Voudouris, 1997] Chris Voudouris. *Guided Local Search for Combinatorial Optimisation Problems*. PhD thesis, Department of Computer Science, University of Essex, 1997.

[Wah and Shang, 1997] Benjamin W. Wah and Yi Shang. Discrete lagrangian-based search for solving MAX-SAT problems. In *IJCAI'97*, pages 378–383, 1997.

[Whigham, 1995] Peter A. Whigham. A schema theorem for context-free grammars. In *1995 IEEE Conference on Evolutionary Computation*, volume 1, pages 178–181, Perth, Australia, 1995. IEEE Press.

[Winston, 1984] Patrick Henry Winston. *Artificial Intelligence*. Addison-Wesley, 1984. ISBN 0-201-08259-4.

[Wolpert and Macready, 1995] David H. Wolpert and William G. Macready. No free lunch theorems for search. Technical Report SFI-TR-95-02-010, The Santa Fe Institute, Sante Fe, NM, USA, 1995.

[Wolpert and Macready, 1997] David H. Wolpert and William G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, April 1997.

[Xing and Zhang, 2004] Zhao Xing and Weixiong Zhang. Efficient strategies for (weighted) maximum satisfiability. In *CP'04*, pages 690–705, 2004.

[Zhang and Stickel, 2000] H. Zhang and M. Stickel. Implementing the Davis-Putnam method. *Journal of Automated Reasoning*, 24(1):277–296, 2000.

[Zhang and Zhang, 1996] Jian Zhang and Hantao Zhang. Combining local search and backtracking techniques for constraint satisfaction. In *AAAI'96*, pages 369–374, 1996.

[Zoltak, 2004] Bartosz Zoltak. VMPC one-way function and stream cipher. In *LNCS 3017: Fast Software Encryption 2004*, pages 210–225. Springer-Verlag, 2004.