Methods of Automatic Algorithm Generation

Stuart Bain, John Thornton, and Abdul Sattar

Institute for Integrated and Intelligent Systems
Griffith University
PMB 50, Gold Coast Mail Centre, 9726, Australia
[s.bain, j.thornton, a.sattar]@griffith.edu.au

Abstract. Many methods have been proposed to automatically generate algorithms for solving constraint satisfaction problems. The aim of these methods has been to overcome the difficulties associated with matching algorithms to specific constraint satisfaction problems. This paper examines three methods of generating algorithms: a randomised search, a beam search and an evolutionary method. The evolutionary method is shown to have considerably more flexibility than existing alternatives, being able to discover entirely new heuristics and to exploit synergies between heuristics.

1 Introduction

Many methods of adapting algorithms to particular constraint problems have been proposed in the light of a growing body of work reporting on the narrow applicability of individual heuristics. A heuristic's success on one particular problem is not an *a priori* guarantee of its effectiveness on another, structurally dissimilar problem. In fact, the "no free lunch" theorems [1] hold that quite the opposite is true, asserting that a heuristic algorithm's performance, averaged over the set of all possible problems, is identical to that of any other algorithm. Hence, superior performance on a particular class of problem is necessarily balanced by inferior performance on the set of all remaining problems.

Adaptive problem solving aims to overcome the difficulties of matching heuristics to problems by employing more than one individual heuristic, or by providing the facility to modify heuristics to suit the current problem. Much of the research into adaptive algorithms has however concerned the identification of which heuristics, from a set of completely specified heuristics, are best suited for solving particular problems. Heuristics in these methods are declared *a priori*, based on the developer's knowledge of appropriate heuristics for the problem domain. This is disingenuous, in that it assumes knowledge of the most appropriate heuristics for a given problem, when the very motivation for using adaptive algorithms is the difficulty associated with matching heuristics to problems.

Our previous work [2] introduced a new representation for constraint satisfaction algorithms that is conducive to automatic adaptation by genetic programming. Additionally, it was demonstrated that from an initial random and poor-performing population, significantly improved algorithms could be evolved.

In this paper we examine other methods to automatically search the space of algorithms possible within this representation. These methods are a beam search, a random search as well as the previously considered evolutionary method.

Existing work on adaptive algorithms will be reviewed in section 2, before the representation to be used in the current experiments is discussed in section 3. The three methods of exploration will be described in section 4, with details of the experiments conducted to evaluate their performance in searching the space of algorithms.

2 Background

A popular paradigm for representing finite domain problems is that of the *constraint satisfaction problem* (CSP). 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. Whilst a multitude of algorithms have been proposed to locate solutions to such problems, this paper focuses on methods that can adapt to the particular problem they are solving. A number of previously proposed adaptive methods will first be discussed.

The MULTI-TAC system proposed by Minton [3, 4] is designed to synthesise heuristics for solving CSPs. Such heuristics are extrapolated from "meta-level theories" i.e. basic theories that describe properties of a partial solution to a CSP. The theories explicated for use with MULTI-TAC lead primarily to variable and value ordering heuristics for complete (backtracking) search. Exploration is by way of a beam search, designed to control the number of candidate heuristics that will be examined. Unlike some of the other adaptive methods, MULTI-TAC is able to learn new heuristics from base theories.

The use of chains of low-level heuristics to adapt to individual problems has also been proposed. Two such systems are the Adaptive Constraint Satisfaction (ACS) system suggested by Borrett et al. [5] and the hyper-heuristic GA system proposed by Han and Kendall [6]. ACS relies on a pre-specified chain of algorithms and a supervising "monitor" function that recognises when the current heuristic is not performing well and directs the search to advance to the next heuristic in the chain. In contrast to a pre-specfied chain, the hyper-heuristic system evolves a chain of heuristics appropriate for a particular problem using a genetic algorithm. Although Borrett exclusively considered complete search methods, their work would allow the use of chains of local search algorithms instead. The same can be said *vice versa* for Han and Kendall's work which considered chains of local search heuristics.

Gratch and Chien [7] propose an adaptive search system specifically for scheduling satellite communications, although the underlying architecture could address a range of similar problems. An algorithm is divided into four seperate levels, each in need of a heuristic assignment. All possibilities for a particular level are searched before a commitment is made to a particular one, and the search proceeds to the next level. In this way, the space of possible methods is pruned and remains computationally feasible. Unfortunately such a method is unable to recognise synergies that may occur between the various levels.

The premise of Nayerek's work [8] 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.

Epstein et al. proposed the Adaptive Constraint Engine (ACE) [9] as a system for learning search order heuristics. ACE is able to learn the appropriate importance of individual heuristics (termed "advisors") for particular problems. The weighted sum of advisor output determines the evaluation order of variables and values. ACE is only applicable for use with complete search, as a trace of the expanded search tree is necessary to update the advisor weights.

With the exception of MULTI-TAC, the primary limitation of these methods is their inability to discover new heuristics. Although ACE is able to multiplicatively combine two advisors to create a new one, it is primarily, like Nayarek's work, only learning which heuristics are best suited to particular problems. Neither [7], which learns a problem-specific conjunctive combination of heuristics, nor [6], which learns a problem-specific ordering of heuristics, actually learn *new* heuristics.

A secondary limitation of the methods discussed (specifically MULTI-TAC and Gratch and Chien's work) is their inability to exploit synergies. Heuristics that perform well in conjunction with other methods, but poorly individually, will not be identified by these two methods. A discussion of synergies is not applicable to the remaining methods, except for the hyper-heuristic GA, where the use of a genetic algorithm permits the identification of synergies. Other factors that should be mentioned include the ability of the methods to handle both complete and local search; the maximum complexity of the heuristics they permit to be learned; and whether the methods are able to learn from failure. The properties of these methods are summarised in the taxonomy of Table 1 below.

Table 1. Taxonomy of Algorithm Adaptation Methods

Name	Learns Local or Complete	Learns New Heuristics	Exploits Synergies	Learns From Failure	Unlimited Complexity	Method of Search
MULTI-TAC	Both	Yes	No	Yes	No	Beam
ACS	Both	No	Yes	No	No	N/A
HHGA	Both	No	Yes	No	No	Evolutionary
Gratch	Both	No	No	Yes	No	Beam
Nayarek	Local	No	Yes	Yes	No	Feedback
ACE	Complete	No	Yes	No	No	Feedback

3 A New Representation for CSP Algorithms

A constraint satisfaction algorithm can be viewed as an iterative procedure that repeatedly assigns domain values to variables, terminating when all constraints are satisfied, the problem is proven unsolvable, or the available computational resources have been exhausted. Both backtracking and local search algorithms can be viewed in this way. The traditional difference between the two methods is that backtracking search instantiates variables only up to the point where constraints are violated, whereas all variables are instantiated in local search regardless of constraint violations. 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?".

Bain et al. [2] proposed a representation capable of handling both complete and local search algorithms, together with a method of genetic programming to explore the space of algorithms possible within the representation. In combination, the representation and genetic programming meet all five criteria discussed in the preceeding section. Although the representation is capable of handling complete search methods, the rest of this paper will concentrate on its use with local search.

Algorithms in this representation are decomposed into three seperate heuristics: the *move contention function*; the *move preference function*; and the *move selection function*. At every iteration, each move (an assignment of a value to a variable) is passed to the *move contention function* to determine which moves will be considered further. For example, we may only consider moves that involve unsatisfied constraints as only these moves offer the possibility of improving the current solution. Each move that has remained in contention is assigned a numeric preference value by the *move preference function*. An example preference function is the number of constraints that would remain unsatisfied for a particular move. Once preference values have been assigned, the *move selection function* uses the preference values to choose one move from the contention list to enact. A number of well-known local search algorithms cast in this representation are shown in Table 2. Extensions for representing a range of more complicated algorithms are discussed in [2].

Table 2. Table of Well-Known Local Search Heuristics

GSAT	{ CONTEND all-moves-for-unsatisfied-constraints; PREFER moves-on-total-constraint-violations; SELECT randomly-from-minimal-cost-moves }
HSAT	{ CONTEND all-moves-for-unsatisfied-constraints; PREFER on-left-shifted-constraint-violations-+-recency; SELECT minimal-cost-move }
TABU	{ CONTEND all-moves-not-taken-recently; PREFER moves-on-total-constraint-violations; SELECT randomly-from-minimal-cost-moves }
WEIGHTING	{ CONTEND all-moves-for-unsatisfied-constraints; PREFER moves-on-weighted-constraint-violations; SELECT randomly-from-minimal-cost-moves }

Table 3. Function and Terminal Sets for Contention

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.
RandomlyTrue :: Integer → Bool	Randomly True Integer percent of the time.
And, Or :: Bool → Bool → Bool	The Boolean AND and OR functions. Definitions as expected.
Not :: Bool	The Boolean NOT function. Definition as expected.
Terminals for use in Contention Heuristics	
Move :: Move	The Move currently being considered.
NumVariables :: Integer	The number of variables in the current problem.
True, False :: Bool	The Boolean values True and False.
10, 25, 50, 75 :: Integer	The integers 0 and 1.

Table 4. Function and Terminal Sets for Preference

Functions for use in Preference Heuristics	
AgeOfMove :: Move → Integer	Returns the number of turns since Move was last taken.
NumWillSatisfy, NumWillUnsatisfy :: Move → Integer	Returns the number of constraints that will be satisfied or unsatisfied by Move, respectively.
Degree :: Move → Integer	Degree returns the number of constraints this Move (variable) affects.
PosDegree, NegDegree :: Move → Integer	Return the number of constraints satisfied by respective variable settings.
DependentDegree, OppositeDegree :: Move → Integer	DependentDegree returns PosDegree if Move involves a currently True variable or NegDegree for a False variable. The reverse occurs for OppDegree.
TimesTaken :: Move → Integer	Returns the number of times Move has been taken.
SumTimesSat, SumTimesUnsat :: Move → Integer	Returns the sum of the number of times all constraints affected by Move have been satisfied or unsatisfied respectively.
SumConstraintAges :: Move → Integer	For all constraints Move participates in, returns the sum of the lengths of time each constraint has been unsatisfied.
NumNewSatisfied, NumNeverSatisfied :: 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 :: Integer → Integer	Returns random value between 0 and Integer-1.
Plus, Minus, Times :: Integer → Integer → Integer	Returns the arithmentic result of its two integer arguments.
LeftShift :: Integer → Integer	Returns its input shifted 16 bits higher.
Terminals for use in Contention Heuristics	
Move :: Move	The Move currently being considered.
NumVariables, NumConstraints :: 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. Function and Terminal Sets for Selection

Functions for use in Selection Heuristics	
RandomFromMax, RandomFromMin, RandomFromPositive, RandomFromAll :: Integer → MoveList → 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	
NumContenders :: Integer	The number of moves in contention.
ListOfMoves :: MoveList	The list of moves determined by the contention stage.
ListOfCosts :: CostList	The list of costs determined by the preference stage.

4 Adapting Algorithms

To study the performance of the three methods, experiments were conducted to evolve algorithms for solving Boolean satisfiability problems. Such problems have been widely studied and have a known hardness distribution. The problem selected (uf100-01.cnf) is taken from the phase-transition region, which is the area where the problems are (on average) the most difficult for traditional backtracking search routines.

4.1 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. With respect to evaluating algorithms, this may be two heuristics that perform poorly individually but excellently together.

To determine whether such synergies occur, a study of possible contention heuristics was conducted using a beam search. The set of possible contention heuristics for the first level of beam search were enumerated from the function and terminal sets shown in Table 3. These heuristics contain at most 1 functional node and are shown in Table 6. As contention heuristics are Boolean functions that determine whether particular moves warrant further consideration, each subsequent level of the beam search will consider more complicated heuristics, by combining additional functional nodes using the Boolean functions: AND, OR and NOT.

As contention heuristics cannot be considered in isolation from preference and selection heuristics, the preference and selection heuristics of the GSAT algorithm were adopted for this experiment. This provides an initial 16 algorithms for evaluation, the results for which are shown in Table 6. Accompanying these are the results for the beam search, which extends the heuristics to all Boolean combinations of up to 2 functional nodes[1]. For a beam width of p, only

[1] with the exception of redundant combinations like “a AND a” and “False OR b”

the heuristics composed entirely from the p best performers are considered, i.e. when the beam width is 2, only heuristics composed of “AgeOverInt(Move, 10)” and “RandomlyTrue(50)” are considered.

Table 6. Beam Search Results

Problem: uf100-01, Tries: 500, Cutoff: 40000								
Heuristics with up to one functional node				Beam search up to two functional nodes				
Rank	Algorithm	Percent Solved	Best 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%

The heuristics examined in the first level of beam search have been delineated into two groups based on the percentage of problems that each was able to solve. Although significant performance improvements can be observed when the better-performing heuristics are combined, the most drastic improvement occurs after the inclusion of one of the poorly-performing heuristics. The “InUnsatisfied(Move)” heuristic, although obvious to human programmers, is not at all obvious to beam search, where its poor individual performance denotes it as a heuristic to be considered later, if at all. Whilst it may be possible to locate good heuristics using beam search, the width of the beam necessary eliminates much of the computational advantage of the method.

4.2 Evolutionary Exploration of the Search Space

Genetic programming [10] has been proposed for discovering solutions to problems when the form of the solution is not known. Instead of the linear (and often fixed length) data structures employed in genetic algorithms, genetic programming uses dynamic, tree-based data structures to represent solutions. The two methods are otherwise quite similar, using equivalent genetic operators to evolve new populations of solutions. When genetic programming is used to evolve algorithms, the data structures are expression trees modelling combinations of heuristics. The fitness function used by the genetic operators relies on solution rates and other performance metrics of the algorithms under test.

Two of the limitations identified from existing work, the inability to exploit synergies and the inability to learn from failure are overcome with genetic programming. Synergies can be exploited as individuals are selected probabilistically to participate in cross-over. Poorly performing individuals still have a possibility of forming part of a subsequent generation. 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. Specifically in local search, information about a candidate algorithm's mobility and coverage [11] can prove useful for comparing 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. No limit is placed on the complexity (size) of the algorithms that may be learned, which will vary depending on the fitness offered by such levels of complexity. Set levels of complexity were an additional limiting factor of some existing work.

Details and results of the experiment can be found in Table 7. These results show that the evolutionary method rapidly evolves good performing algorithms from an initially poor performing population. Although the experiment was continued for 100 generations, there was little improvement after generation 30.

Table 7. Conditions and Results for the Genetic Programming Experiment

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 number of moves required		90	90.56%	1.35	876	876
by the state-of-the-art [12]	594	99	92.88%	1.73	1070	876

4.3 Random Exploration of the Search Space

In order to demonstrate that the observed performance improvements in the evolutionary experiment over time are not purely the result of fortuitously generated algorithms, the experiment was repeated without the genetic operators. That is, each generation of the population was composed entirely of randomly generated elements. 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 tabulated in Table 8, when three different (practical) limits are placed on the

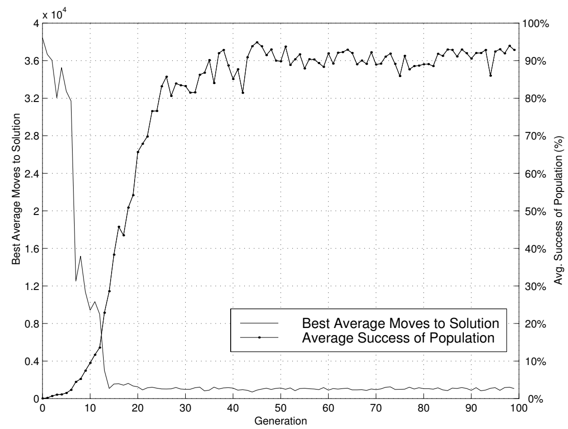

Fig. 1. Results for the genetic programming experiment

size of the generated contention and preference trees[2]. 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.

Table 8. Results for the Random Exploration Experiment

	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

5 Conclusions and Future Work

This paper has demonstrated that within the space of algorithms, synergies do exist between heuristics, so a heuristic that performs poorly individually may perform well in conjunction with other heuristics. For this reason, beam search is not the most appropriate method for searching the space of algorithms.

Furthermore, the usefulness of genetic programming was demonstrated by comparing it with an entirely random method of search. As genetic programming begins with a similar, entirely random set of solutions, the observed performance

[2] Selection heuristics are restricted by the function and terminals sets to have exactly 4 nodes.

improvements are attributable to the genetic operators. Even with a fixed set of functions and terminals, albeit one large enough to be combined in many novel ways, an initial random and poorly-performing population of algorithms was significantly improved by the application of genetic programming operating within a recently proposed representation.

6 Acknowledgments

The authors would like to acknowledge the support of the Australian Research Council Large Grant A00000118 in conducting this research.

References

1. Wolpert, D.H., Macready, W.G.: No free lunch theorems for optimization. IEEE Transactions on Evolutionary Computation **1** (1997) 67–82
2. Bain, S., Thornton, J., Sattar, A.: Evolving algorithms for constraint satisfaction. In: 2004 Congress on Evolutionary Computation, Portland, Oregon (2004) To Appear.
3. Minton, S.: An analytic learning system for specializing heuristics. In: IJCAI '93: Proceedings of the 13th International Joint Conference on Artificial Intelligence, Chambéry, France (1993) 922–929
4. Minton, S.: Automatically configuring constraint satisfaction programs: A case study. Constraints **1** (1996) 7–43
5. Borrett, J.E., Tsang, E.P.K., Walsh, N.R.: Adaptive constraint satisfaction: The quickest first principle. In: European Conference on Artificial Intelligence. (1996) 160–164
6. Han, L., Kendall, G.: An investigation of a Tabu assisted hyper-heuristic genetic algorithm. In: 2003 Congress on Evolutionary Computation. Volume 3., IEEE Press (2003) 2230–2237
7. Gratch, J., Chien, S.: Adaptive problem-solving for large-scale scheduling problems: A case study. Journal of Artificial Intelligence Research **1** (1996) 365–396
8. Nareyek, A.: Choosing search heuristics by non-stationary reinforcement learning. In: M.G.C. Resende and J.P. de Sousa (Eds), Metaheuristics: Computer Decision Making, Kluwer Academic Publishers (2001) 523–544
9. Epstein, S.L., Freuder, E.C., Wallace, R., Morozov, A., Samuels, B.: The adaptive constraint engine. In Hentenryck, P.V., ed.: CP '02: Principles and Practice of Constraint Programming. (2002) 525–540
10. Koza, J.: Genetic Programming: On the programming of computers by means of natural selection. MIT Press, Cambridge, Massachusetts (1992)
11. Schuurmans, D., Southey, F.: Local search characteristics of incomplete SAT procedures. Artificial Intelligence **132** (2001) 121–150
12. Hutter, F., Tompkins, D., Hoos, H.: Scaling and probabilistic smoothing: Efficient dynamic local search for SAT. In: CP '02: Principles and Practice of Constraint Programming, Springer Verlag (2002) 233–248