# Evolving Algorithms for Constraint Satisfaction

Stuart Bain and John Thornton and Abdul Sattar
Institute for Integrated and Intelligent Systems
Griffith University
PMB 50, Gold Coast Mail Centre, 9726, Australia
Email: [s.bain, j.thornton, a.sattar]@griffith.edu.au

*Abstract*— **This paper proposes a framework for automatically evolving constraint satisfaction algorithms using genetic programming. The aim is to overcome the difficulties associated with matching algorithms to specific constraint satisfaction problems. A representation is introduced that is suitable for genetic programming and that can handle both complete and local search heuristics. In addition, the representation is shown to have considerably more flexibility than existing alternatives, being able to discover entirely new heuristics and to exploit synergies between heuristics. In a preliminary empirical study, it is shown that the new framework is capable of evolving algorithms for solving the well-studied problem of boolean satisfiability testing.**

## I. Introduction

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. 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 these difficulties by employing more than one individual heuristic, or by providing the facility to modify heuristics to suit the current problem. More generally, an adaptive system can be considered to embody a space of possible heuristics. As the search progresses, information gathered about the structure of the problem or the efficacy of the various heuristics, is used by the system to explore the space of possible heuristics and locate the one most applicable to the current problem. In addition to overcoming the limitations imposed by a single heuristic, an adaptive system removes the need for the developer to determine the most appropriate heuristic beforehand.

Despite this, much of the research into adaptive algorithms has 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.

Existing work on adaptive algorithms will be discussed in section II, before a new representation that overcomes these difficulties is presented in section III. An example of its use, and how it has been extended with compound heuristics, will be presented in sections IV and V respectively. How the search space of algorithms may be explored and expanded is described in sections VI and VII, followed by the presentation of experimental results in section VIII.

## II. Background

One paradigm that has proven particularly popular 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 [2], [3] 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. [4] and the hyper-heuristic GA system proposed by Han and Kendall in [5]. 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 [4] exclusively considered complete search methods, their work

would preclude the use of chains of local search algorithms instead. The same can be said *vice versa* for [5] which considered chains of local search heuristics.

Gratch and Chien [6] 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 [7] 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.

In [8], Epstein et al. proposed the Adaptive Constraint Engine (ACE) 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 [6], which learns a problem-specific conjunctive combination of heuristics, nor [5], which learns a problem-specific ordering of heuristics, actually learns *new* heuristics.

A secondary limitation of the methods discussed (specifically those of [3], [6]) is their inability to exploit synergies. Heuristics that perform well in conjunction with other methods, but poorly individually, will not be identified by the two methods. A discussion of synergies is not applicable to the remaining methods, except for [5] 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. How the new representation and genetic programming will address these points will be discussed in the following sections.

## III. 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. How the values and the variables are chosen depends on the heuristics of the particular algorithm. Such a heuristic algorithm can be defined in the new representation by the specification of three functions: the *move contention function*; the *move preference function*; and the *move selection function*.

```
ALGORITHM {
    CONTEND some-moves-to-consider;
    PREFER moves-according-to-some-metric;
    SELECT one-move-to-enact
}
```

Fig. 1. Representation of a constraint algorithm

Each move is passed in sequence to the *move contention function* to determine which moves (assignments of values to variables) are to be further considered by the search algorithm. Examples of this type of function are: "all moves that involve unsatisfied constraints"; "all moves that haven't been taken recently"; or "all moves involving unassigned variables". The resultant list of moves is then passed one move at a time to the move preference function, which assigns a numeric preference value to each move. Examples of preference functions include: "the count of unsatisfied constraints"; "the time since this move was last taken"; or "the maximally constrained variable". Once preference values have been assigned, the *move selection function* uses the preference values to choose one move from the contention list to enact. Some commonly used selection functions are "a random selection from the best moves" and "a random selection from improving moves".

The example functions mentioned have been drawn from both the local and complete search domains to demonstrate that that the proposed representation is applicable to both types of search. Both backtracking and local search algorithms for constraint satisfaction can be viewed as iteratively assigning values to variables. 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. As backtracking maintains a complete record of its search, it is capable of exploring the entire search space. Local search routines rarely record so much information, selecting a promising new solution from the neighbourhood of the current solution on the basis of a heuristic function.

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?". Although the representation is capable of handling complete search algorithms, overcoming one of the limitations of some existing

work, the rest of the paper will concentrate on its use for local search methods.

## IV. The Representation in Operation

A demonstration of the representation in operation will be presented using the well-known GSAT algorithm [9] and reference to a small graph colouring problem. GSAT was selected because it is widely known, relatively simple, and was instrumental in the development of local search for constraint satisfaction. Figure 2 presents the GSAT algorithm in the new representation and the graph colouring problem is shown in Figure 3.

```
GSAT {
    CONTEND all-moves-for-unsatisfied-constraints;
    PREFER moves-on-total-constraint-violations;
    SELECT randomly-from-minimal-cost-moves
}
```

Fig. 2. Representation of the GSAT algorithm

The aim of the problem is to find an assignment of colours to each of the three countries, such that no countries sharing a border are coloured the same. For this problem, there are 3 variables (the three countries); 3 domain values for each variable (the colours black, gray and white); and 3 constraints (for the three contiguous borders). Local search is being considered, so the problem begins with a random instantiation of values to variables that has resulted in one constraint violation, as shown in Figure 3.

### A. Step 1 - Move Contention

Move contention determines which moves are currently available for the search algorithm. All moves are considered, being passed in sequence to the GSAT heuristic that returns True if (and only if) the move involves a variable in an unsatisfied constraint. In this case, this heuristic returns True for the variables $B$ and $C$. The possible moves are:

1) $B \leftarrow Black$
2) $B \leftarrow White$
3) $C \leftarrow Black$
4) $C \leftarrow White$

These potential moves are now passed to the move preference stage of the algorithm.

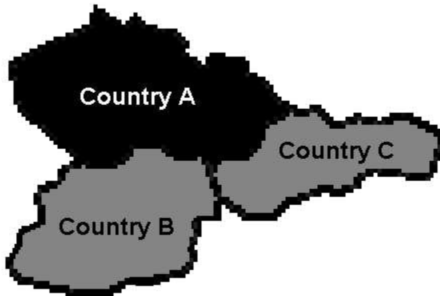


Fig. 3. A graph colouring problem

### B. Step 2 - Move Preference

Move preference involves assigning to each of the contending moves a numeric value representing how well that move satisfies a particular metric. GSAT ranks moves according to the total number of constraints that would be unsatisfied if each move was taken. The number of constraints that would be unsatisfied for each of the four contending moves is as follows:

1) $B \leftarrow White = 0$
2) $C \leftarrow White = 0$
3) $B \leftarrow Black = 1$
4) $C \leftarrow Black = 1$

The moves and their preference values are now passed to the final stage of the algorithm.

### C. Step 3 - Move Selection

Move selection uses the results of the preference stage to select a move to enact. GSAT makes a random selection from amongst the (two) best moves, which in this case, both lead to a satisfying solution. After enacting this move, the problem is solved and the algorithm terminates.

## V. Compound Heuristics

A number of published local search algorithms can be posed in the representation as it has been described so far. A selection of these algorithms are listed in Table I. Some algorithms however, are not able to be posed in such simple terms. For this reason, three compound heuristics are presented: *probabilistic choice*; *possibilistic choice*; and *comparative choice*.

### A. Probabilistic choice

Probabilistic choice is employed by the WSAT heuristic [10]. Instead of applying a single strategy, probabilistic choice allows two seperate algorithms to be used. One of the two algorithms is selected probabilistically for use each turn. This introduces an element of randomness that may assist a search in escaping a local minimum.

### B. Possibilistic choice

Possibilistic choice occurs in a number of algorithms (such as DLM [11]), that repeatedly use a single algorithm until it is no longer able to be applied. As an example, the search may operate in a greedy manner until no improving moves exist. When an improving move is no longer possible, the algorithm may begin making random or cost neutral moves that would not normally be accepted. Possibilistic choice requires three algorithms as arguments, only if the first algorithm is possible is the second algorithm enacted. Otherwise the third algorithm will be used. There is no restriction that all algorithms must be distinct.

### C. Comparative choice

Comparative choice has been used by the Novelty [12] family of algorithms, and references four different algorithms as arguments. The moves that would be taken by the first two specified algorithms are compared, in the case of Novelty, these are the “best” move and the most “recent” move algorithms. If both algorithms would make the same move, the third algorithm is enacted, otherwise the fourth algorithm would be used.

In Novelty, the CONTEND heuristic requires the selection of a random constraint (see Table I), which recurs throughout the algorithm. This creates an ambiguity: either each instance of CONTEND randomly picks its own constraint or a single randomly selected constraint applies to all instances (for Novelty, the same constraint does apply to all instances). This ambiguity could be removed by enforcing the rule that all duplications of a given heuristic (within the same algorithm) become exact copies of a single underlying heuristic. Hence, if a random choice is made, it will be the same for all versions. Alternatively, the expression trees could be treated as directed acyclic graphs, so that only particular duplications are equivalent[1]. An algorithm would not be restricted to a single instance of each heuristic, allowing for cases where different evaluations might be desired. Determining which of the two methods is most appropriate is an ongoing area of research.

<table>
<tr><td>GSAT</td><td>{ CONTEND all-moves-for-unsatisfied-constraints;<br>PREFER moves-on-total-constraint-violations;<br>SELECT randomly-from-minimal-cost-moves }</td></tr>
<tr><td>HSAT</td><td>{ CONTEND all-moves-for-unsatisfied-constraints;<br>PREFER on-left-shifted-constraint-violations-+-recency;<br>SELECT minimal-cost-move }</td></tr>
<tr><td>TABU</td><td>{ CONTEND all-moves-not-taken-recently;<br>PREFER moves-on-total-constraint-violations;<br>SELECT randomly-from-minimal-cost-moves }</td></tr>
<tr><td>WEIGHT-ING</td><td>{ CONTEND all-moves-for-unsatisfied-constraints;<br>PREFER moves-on-weighted-constraint-violations;<br>SELECT randomly-from-minimal-cost-moves }</td></tr>
<tr><td>WSAT</td><td>{ PROBABILISTIC<br>{ CONTEND all-moves-for-a-random-constraint;<br>PREFER moves-on-new-constraint-violations;<br>SELECT randomly-from-minimal-cost-moves };<br>{ CONTEND all-moves-for-a-random-constraint;<br>PREFER moves-on-new-constraint-violations;<br>SELECT randomly-from-all-contenders }</td></tr>
<tr><td>NOVELTY</td><td>{ COMPARATIVE {<br>IF { CONTEND all-moves-for-a-random-constraint;<br>PREFER moves-on-total-constraint-violations;<br>SELECT randomly-from-minimal-cost-moves }<br>== { CONTEND all-moves-for-a-random-constraint;<br>PREFER moves-on-recency-of-move;<br>SELECT randomly-from-minimal-cost-moves };<br>THEN { PROBABILISTIC {<br>{ CONTEND all-moves-for-a-random-constraint;<br>PREFER moves-on-total-constraint-violations;<br>SELECT randomly-from-minimal-cost-moves };<br>{ CONTEND all-moves-for-a-random-constraint;<br>PREFER moves-on-total-constraint-violations;<br>SELECT from-second-lowest-cost-moves } };<br>ELSE { CONTEND all-moves-for-a-random-constraint;<br>PREFER moves-on-total-constraint-violations;<br>SELECT randomly-from-minimal-cost-moves }<br>} }</td></tr>
</table>

TABLE I
TABLE OF WELL-KNOWN LOCAL SEARCH HEURISTICS

## VI. ADAPTING ALGORITHMS

The aim in developing a new representation has been to provide a framework for the adaptation and discovery of new algorithms. One of the underlying premises of this work is that a representation capable of expressing the diverse range of current algorithms, without dictating their explicit definition, implies a suitable level of complexity for new algorithms. Preceding sections have demonstrated that the representation is capable of handling the level of complexity present in existing algorithms.

One method that has been proposed for discovering solutions when the form of the solution is not pre-determined (or is unknown) is genetic programming [13]. Genetic programming uses a dynamic, tree-based data structure to overcome the limitation of the linear (and often fixed length) data structures used by genetic algorithms. As a brief example of genetic programming, consider regressing some data to a linear function, $mx + c$. Providing that the data is from a linear function, regression requires the determination of values for $m$ and $c$. Using a genetic algorithm, $m$ and $c$ would entirely comprise the chromosome, and we are limited to discovering equations of a linear form. Alternatively, a genetic program for the same purpose would allow the same two constants $m$ and $c$ but possibly more, the argument $x$ and the operators $+$ & $\times$ to form part of the solution. These are combined by genetic programming into an expression tree representing the function to be learned. Where the genetic algorithm is restricted to learning first-order polynomials, this method has the advantage of being able to learn a range of polynomial functions.

Adaptation of a genetic program takes place using methods analagous to those employed in a genetic algorithm for selection, cross-over and mutation, except that the GP variants are designed to operate on trees, rather than a string of symbols. An example of a heuristic represented as an expression tree is diagrammed in Figure 4. The selection operation is very similar, as the fitness of tentative solutions must still be evaluated to determine which solutions will beget the next generation. Cross-over operates differently than in the standard GA, selecting a random subtree from each of two parent solutions and interchanging them to create two new solutions. Finally, mutation in a genetic program creates a new solution by replacing an existing subtree with a new, randomly-generated one.

Genetic programming addresses the remaining limitations identified in existing work. Synergies can be exploited, as individuals are not removed from the population as a result of poor performance. As individuals are selected probabilistically to participate in cross-over, individuals that have performed poorly on their own may still form part of a subsequent

[1]The authors would like to acknowledge the input of Peter Stuckey, University of Melbourne, for this suggestion.

generation. The other limitation of some existing approaches is their inability to learn from failure. Even if an algorithm does not locate any solutions, information about how close it came to solutions or how much of the search space it explored can form part of a fitness function.

Algorithms in the proposed representation are adapted by genetic programming operations that must preserve the syntactic structure of the original algorithm. This is easily accomplished with both the cross-over and mutation operations. In cross-over, only similarly-typed structures from two parents may be interchanged. All functions and terminals that compose the three different heuristics also have associated types that must be preserved. This is always preserved by any genetic operation that operates on an algorithm.

The algorithms listed in Table I were described using easily recognisable English descriptions of their heuristics. If the genetic operators were unable to decompose heuristics any further than these descriptions, it could still explore a bounded space of algorithms simply by interchanging the component heuristics. The aim of this work is to develop new heuristics by combining individual functions and terminals in novel ways. This requires that heuristics be broken down into their component parts. The space of heuristics can then be expanded by combining these functions and terminals with a number of generic functions. The more detailed expressions behind the English description, will now be presented for the GSAT algorithm as an example[2].

Contention in GSAT passes every move currently possible to the "InUnsatisfied" function, which returns boolean True if the move will be in contention or False if the move is not to be considered further. This function can be expressed in more detail as: "num-constraints-that-would-be-satisfied(move)>0".

The GSAT algorithm prefers "moves-on-total-constraint-violations". Preference assigns to each move in the contention list a numeric value, which in the case of GSAT, is the difference between the number of constraints that will become satisfied by a particular move and the number of constraints that will become unsatisfied. The expression for this function is "NumWillSatisfy(move) - NumWillUnsatisfy(move)". It is shown as an expression tree in Figure 4.

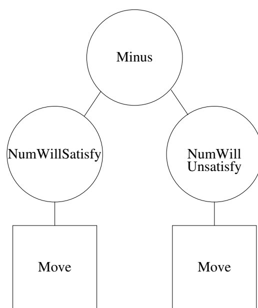


Fig. 4. The GSAT preference heuristic as an expression tree

[2]Readers may at this stage wish to refer to the tables of functions and terminals for the three component heuristics, which are presented in Tables IV-VI at the end of this paper.

The selection heuristics have not yet been modelled in sufficient detail that they could be adapted through cross-over or mutation. However, four different selection heuristics are listed in Table V. Selection heuristics can still be represented as expression trees, only the set of functions and terminals has currently been limited, such that only a fixed set of heuristics is possible.

## VII. Learning New Heuristics

The previous section has described how heuristics may be represented as expression trees that can then be adapted with genetic programming. Although the set of functions and terminals is fixed for all heuristics, the presence of a mixture of functions (such as "AND", "OR" and "NOT" for contention and "PLUS", "MINUS" and "TIMES" for preference) permit a range of new heuristics to be learned, as well as existing heuristics to be combined in novel ways. 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. The ability of this representation to learn new algorithms and without an *a priori* complexity bound address the two remaining limitations identified from the literature.

It is believed that a further level of "granularity" may be exploited, where appropriate functions and terminals are not specified, but are instead learned from a much more finely grained meta-knowledge of a constraint system. This is beyond the scope of the current paper, but will form the basis of our future work into evolving algorithms. The experimental study presented in the next section will be limited to the function and terminal sets tabulated in Tables IV, V & VI.

## VIII. Experimental Study and Results

An initial experimental study has been conducted to test the ability of the new representation and genetic programming to successfully evolve heuristic expression trees. This study is the precursor to a more detailed and extensive study to be presented in future work. Performing multiple runs of a constraint algorithm is a time-consuming task, and adapting algorithms with a method such as genetic programming is even more so, as the performance of an entire population of algorithms must be evaluated. The challenges facing an evolutionary algorithm with a computationally intensive fitness function cannot be overstated. It is not the authors' intention to demonstrate that this method can immediately generate algorithms competitive with state-of-the-art constraint satisfaction algorithms. The object of this exploratory study is to demonstrate that from a random starting population, an evolutionary technique can improve a population of algorithms' performance over time, within the framework of the presented representation.

Algorithm performance is most often presented in the constraint literature in terms of the number of moves required to find a solution or in terms of the time taken to locate a solution. Results are averaged over a number of attempts, as many heuristic algorithms employ an element of randomness when navigating the search space as well as randomly initialising their solution vector. This randomness makes the algorithms

non-deterministic, and a single, possibly "lucky", run is not necessarily indicative of an algorithm's performance. For this reason, figures presented for the time or number of moves required to locate a solution are averaged over a number of trials.

| **Population Composition** | |
|---|---|
| Population Size | 100 |
| Copied from previous generation | 25 |
| Randomly selected and crossed-over | 70 |
| New elements generated | 5 |
| Elitism | Yes |
| **Evaluation of Algorithm Fitness** | |
| $F = Standardised(UnsatConstraints_i) + 100 * SuccessRate_i$ | |
| Test Problem | uf20-01.cnf |
| Number of runs for each algorithm | 25 |
| Maximum moves per run | 10000 |
| Number of moves required by state-of-the-art | 24[4] |
| Number of generations | 75 |

TABLE II
EXPERIMENT CONDITIONS

An initial random population of algorithms was generated and the fitness of each member evaluated. One of the fitness measures being used is the fewest constraint violations that the algorithm was able to achieve during its search. This is averaged over all runs of an algorithm and then standardised (in the usual genetic programming sense) to the worst performing element of the population. The second figure is the number of times that the algorithm found a solution. scaled between (1-100), which rapidly becomes the dominant part of the fitness function, as the problemhas less that 100 constraints that to be unsatisfied. This fitness functionis presented in Table II. The constraint problemselected for this test is a randomly generated satisfiability problem with 20 variables and 91 constraints, available from the SATLIB benchmark set[3]. The maximum number of moves (per run) that each algorithm is given is well in excess of the number required by state-of-the-art algorithms[4].

The three heuristics that compose each algorithm are not permitted to "inter-breed". That is, contention heuristics may only be crossed with other contention heuristics and so on. Which of the three heuristics are crossed is determined randomly, and more than one of the heuristics may be crossed in a single cross-over operation. Although new selection heuristics cannot be learned due to the limited set of functions and terminals specified, cross-over can swap selection heuristics between two algorithms, pairing each selection heuristic with a different contention and preference combination.

Experimental results are shown in Table III, Figure 5 (average success of the population) and in Figure 7 (the best performance of any individual algorithm). The use of elitism to convey the best performing elements of one generation to the next would generally limit the amount of noise present in the results, and provide a monotonically increasing "best" fitness. In these results however, the evaluation of each algorithm from generation to generation, is in itself a stochastic process. Only a relatively small number of runs was performed which is insufficient to remove all of the noise introduced by the inherent randomness of the algorithms.

[3]http://www.satlib.org

[4]Determined using the SAPS weighting algorithm of [14]

| Generation | Mean Success | Mean Unsatisfied Constraints | Best Turns to Solution | Total Diversity (Percent) |
|---|---|---|---|---|
| Problem: uf20-01.cnf | | | | |
| 0 | 11.4% | 5.01 | 374 | 100% |
| 9 | 66.3% | 1.37 | 166 | 61.2% |
| 19 | 84.4% | 0.97 | 136 | 67.3% |
| 29 | 85.9% | 0.41 | 103 | 71.4% |
| 39 | 84.5% | 0.55 | 76 | 59.2% |
| 49 | 87.2% | 0.54 | 60 | 59.2% |
| 59 | 90.9% | 0.46 | 60 | 44.9% |
| 69 | 94.2% | 0.22 | 54 | 46.9% |
| 74 | 89.1% | 0.35 | 61 | 55.1% |

TABLE III
RESULTS OF ALGORITHM EVOLUTION

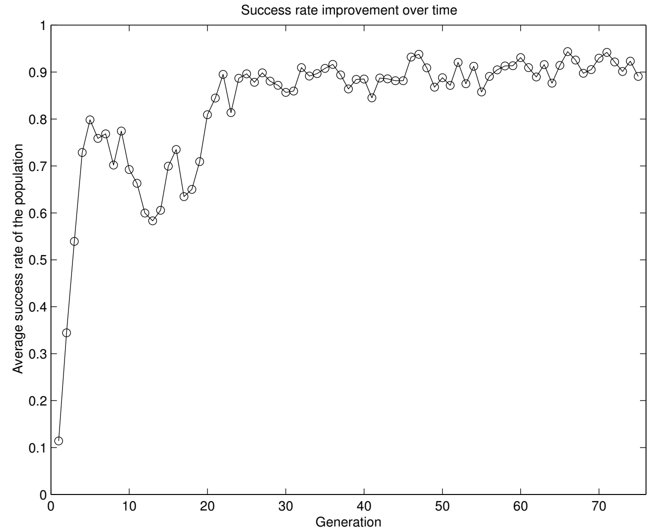


Fig. 5. Success rate improvement over time

Despite occasional decreases in the best fitness of the population, a steady improvement in performance can be observed up until convergence occurred at around generation 50. At this point, the population had converged to a very small subset of the available functions and terminals. A total of 49 functions and terminals were presented in Tables IV-VI. The best performing algorithms used only combinations of the "InRandom", "WontUnsatisfy" and "Or" contention functions, with either "NumWillSatisfy" or "SumConstraintAges" as a preference function and either "RandomFromMaxmimum" or "RandomFromPositive" used for selection. Only the contention functions tended to grow significantly, with sizable trees (more than 50 nodes each) composed solely of the three contention functions mentioned. However, the nature of the "OR" function means that any further expansion beyond "InRandom OR WontUnsatisfy" does not alter the

operation of the heuristic. The cumulative genetic diversity of the population is shown for both the initial and final generations in Figure 6. As can be seen from this figure, most of the best performing individuals in the final population were drawn from a very small subset of the available functions and terminals.

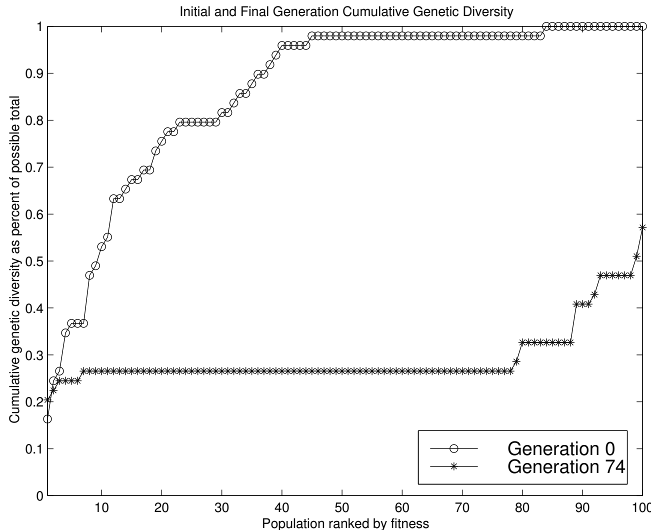


Fig. 6. Cumulative population diversity for initial and final generation

The performance of the best individual algorithm improved by a factor of 6 between generation 0 and generation 74. During the same period, the average success rate of the population had risen from 11% to over 90%. Best performance is the more important measure, however as this reflects an undeniable improvement in the system, whereas average performace can be increased simply by removing poorly performing individuals and does not necessarily reflect an improvement in individual algorithms. The lowest number of moves required to locate a solution is plotted against generation in Figure 7. From this diagram it is clear that even from a relatively poor performing initial start, the proposed algorithm representation used with genetic programming allows significantly improved algorithms to be evolved.

## IX. Conclusions and Future Work

This paper has introduced a new representation for constraint satisfaction algorithms that can model both complete and local search methods. It has further shown how the application of genetic programming to the new representation can evolve effective constraint solving algorithms, while at the time time addressing some of the limitations of existing methods (such as learning new algorithms and exploiting synergies).

Future work will concentrate on a much larger experimental study; the inclusion of compound heuristics in the genome; and a determination of the most appropriate method of handling multiple instances of a single heuristic. Even with a fixed set of functions and terminals, albeit one large enough to be combined in many novel ways, a random initial and

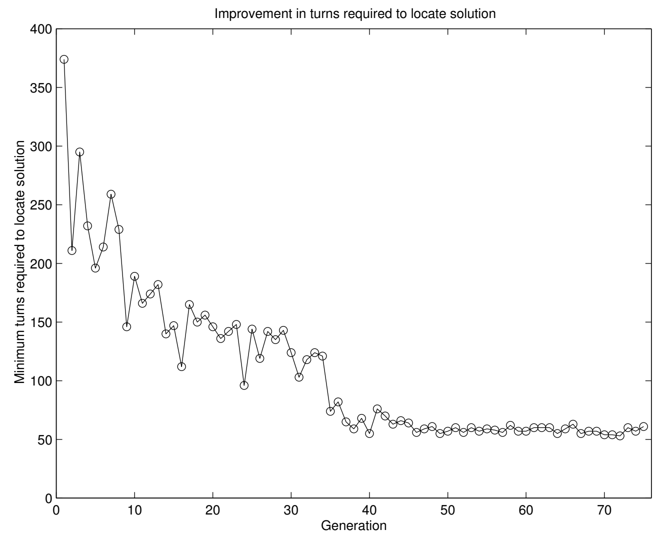


Fig. 7. Minimum turns required to locate a solution

poor-performing population of algorithms was significantly improved by the application of genetic programming operating within the presented representation.

## X. Acknowledgment

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

## References

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

[2] Steven Minton. An analytic learning system for specializing heuristics. In *IJCAI '93: Proceedings of the 13th International Joint Conference on Artificial Intelligence*, pages 922–929, Chambéry, France, 1993.

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

[4] 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.

[5] 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.

[6] Jonathan Gratch and Steve Chien. Adaptive problem-solving for large-scale scheduling problems: A case study. *Journal of Artificial Intelligence Research*, 1:365–396, May 1996.

[7] Alexander Nareyek. Choosing search heuristics by non-stationary reinforcement learning. In *M.G.C. Resende and J.P. de Sousa (Eds), Metaheuristics: Computer Decision Making*, pages 523–544. Kluwer Academic Publishers, 2001.

[8] Susan L. Epstein, Eugene C. Freuder, Richard Wallace, Anton Morozov, and Bruce Samuels. The adaptive constraint engine. In Pascal Van Hentenryck, editor, *CP '02: Principles and Practice of Constraint Programming*, pages 525–540, 2002.

[9] Bart Selman, Hector J. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. In Paul Rosenbloom and Peter Szolovits, editors, *AAAI'92*, pages 440–446, Menlo Park, California, 1992. AAAI Press.

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

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

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

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

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


| **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. Definition as expected | |
| **Not** | Bool → Bool |
| The boolean NOT function. True iff its input is False. | |
| **Terminals for use in Contention Heuristics** | |
| **Move** | Move |
| The move currently being considered for contention. | |
| **NumVariables** | 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 IV

FUNCTION AND TERMINAL SETS FOR CONTENTION

| **Functions for use in Selection Heuristics** | |
|---|---|
| **RandomFromMaximum** | ListOfMoves → ListOfCosts → Move |
| A random selection from the maximum cost moves. | |
| **RandomFromMinimum** | ListOfMoves → ListOfCosts → Move |
| A random selection from the minimum cost moves. | |
| **RandomFromPositive** | ListOfMoves → ListOfCosts → Move |
| A random selection from positive cost moves. | |
| **RandomFromAll** | ListOfMoves → ListOfCosts → Move |
| A random selection from all moves | |
| **Terminals for use in Selection Heuristics** | |
| **ListOfMoves** | ListOfMoves |
| The list of moves determined by the Contention stage. | |
| **ListOfCosts** | ListOfMoves |
| The accompanyng list of costs determined by the Preference phase. | |

TABLE V

FUNCTION AND TERMINAL SETS FOR SELECTION

| **Functions for use in Preference Heuristics** | |
|---|---|
| **AgeOfMove** | Move → Integer |
| Returns the number of turns since Move last taken. | |
| **NumWillSatisfy** | Move → Integer |
| Returns the number of constraints that will be satisfied by Move. | |
| **NumWillUnsatisfy** | Move → Integer |
| Returns the number of constraints that will be unsatisfied by Move. | |
| **Degree** | Move → Integer |
| Since each Move concerns a single variable, Degree returns the number of constraints this Move (variable) participates in. | |
| **PosDegree** | Move → Integer |
| As for Degree, but returns the total number of constraints that are satisfied by a setting of True for this variable. | |
| **NegDegree** | Move → Integer |
| Opposite of PosDegree. | |
| **DependantDegree** | Move → Integer |
| Returns the total number of constraints that this variable participates in that are additionally satisfied by its current value. | |
| **OppositeDegree** | Move → Integer |
| As for DependantDegree but for constraints not satisfied by the variable's current value. | |
| **TimesTaken** | Move → Integer |
| Returns the number of times Move has been taken. | |
| **SumTimesUnsatisfied** | Move → Integer |
| Returns the sum of the the number of times all constraints Move affects, have been unsatisfied. | |
| **SumTimesSatisfied** | Move → Integer |
| As above, but for the number of times Satisfied. | |
| **SumConstraintAges** | Move → Integer |
| For all constraints Move participates in, returns the sum of the lengths of time each constraint has been unsatisfied | |
| **NumNewSatisfied** | Move → Integer |
| Returns the number of constraints not currently satisfied that will be satisfied by Move. | |
| **NumNeverSatisfied** | Move → Integer |
| Returns the number of constraints that have never been satisfied that Move will satisfy. | |
| **RandomValue** | Integer → Integer |
| Returns a random value between 0 and its input-1. | |
| **Plus** | Integer → Integer → Integer |
| Returns the sum of its two input arguments. | |
| **Minus** | Integer → Integer → Integer |
| Returns the subtraction of its two input arguments. | |
| **Times** | Integer → Integer → Integer |
| Returns the product of its two input arguments. | |
| **LeftShift** | Integer → Integer |
| Returns its input shifted 16 bits higher. | |
| **Terminals for use in Contention Heuristics** | |
| **Move** | Integer → Integer |
| The move currently being considered for preference. | |
| **NumVariables, NumConstraints** | Integer → Integer |
| The number of variables and constraints in the current problem. | |
| **NumFlips** | Integer → Integer |
| The number turns that have already passed. | |
| **0, 1** | Integer → Integer |
| The integers 0 and 1. | |

TABLE VI

FUNCTION AND TERMINAL SETS FOR PREFERENCE