A Comparison of Evolutionary Methods for the Discovery of Local Search Heuristics

Stuart Bain, John Thornton, and Abdul Sattar

Institute for Integrated and Intelligent Systems
Griffith University
PMB 50, GCMC 9726, Australia
{s.bain, j.thornton, a.sattar}@griffith.edu.au

Abstract. Methods of adaptive constraint satisfaction have recently become of interest to overcome the limitations imposed on "black-box" search algorithms by the no free lunch theorems. Two methods that each use an evolutionary algorithm to adapt to particular classes of problem are the CLASS system of Fukunaga and the evolutionary constraint algorithm work of Bain et al. We directly compare these methods, demonstrating that although special purpose methods can learn excellent algorithms, on average standard evolutionary operators perform even better, and are less susceptible to the problems of bloat and redundancy.

1 Introduction

Problems involving constraints are ubiquitous in both theoretical and applied computer science, and as many real-world problems can be easily formulated as boolean satsifiability problems, the development of algorithms for satisfiability testing remains an important area of research. Consider however, the problem of choosing the most appropriate algorithm for a particular class of constraint problems. Are algorithms with favourable benchmark performance most suited to the specific problems of interest?

According to the no free lunch theorems [1], the short answer to this question is "no". Not only is an algorithm's performance on some problem classes not indicative of its performance on others, unless the algorithm exploits some hidden structure of a problem class then above-average performance on that class must necessarily be balanced by below-average performance on other classes. Given this, the challenge becomes the development and matching of the most appropriate algorithm for a particular class of problem.

As algorithm development is a time-consuming task, methods of adaptive problem solving that can automatically adapt to the specific problems of interest, without the necessity of human problem-solving experience, have started to receive attention [2–4]. One method that has proved successful is to treat constraint algorithms as expressions to be modified by some form of evolutionary procedure. Such a procedure applies one or more genetic operators to the "fittest" algorithms in order to breed the next generation of algorithms.

The contribution of this paper is a direct comparison of the various genetic operators that may be used to breed successive generations of algorithms.

2 Evolving Algorithms

When considering methods of algorithm adaptation, a number of features are crucial to the expressiveness and performance of an adaptive system [5]. 1) The ability to represent both complete and local search routines; 2) Unrestricted complexity; 3) Appropriateness for satisfiable and over-constrained problems; 4) The ability to learn from failure; and 5) The ability to recognise and exploit synergies, subexpressions that perform well together but poorly individually.

A constraint satisfaction algorithm can be viewed as an iterative procedure that repeatedly assigns (in the case of complete search) or reassigns (in the case of local search) domain values to variables. Procedures of either type usually rank potential variable-value instantiations according to heuristic merit. This heuristic can be viewed as an expression composed from functions and terminals (constants) describing the nature of the constraint problem and the state of the search. The specific functions and constants used will determine whether the possible expressions lead to heuristics suited for complete or local search. Provided that a sufficiently expressive set of functions is chosen, such expressions do not place an *a priori* bound on the complexity of possible heuristics.

An evolutionary method of adaptation exhibits characteristics 3, 4 and 5 above. Evolutionary algorithms rely on a measure of fitness to rank candidate algorithms, and can include metrics appropriate to satisfiable problems (such as success rate); or to over-constrained problems (such as best solution cost); and that distinguish between algorithms even when they fail to find a solution (such as measures of depth, mobility and coverage for local search [6]).

It is not suprising then, that in a number of previous works evolutionary techniques have been applied to the problem of automatically adapting constraint satisfaction algorithms, specifically, the CLASS system of Fukunaga [7, 8] and the evolutionary constraint algorithm work of Bain et al. [4, 5].

CLASS constructs algorithms composed mainly of *if..then* style production rules and relies heavily on measures that have been used in the GSAT, WALKSAT and NOVELTY families. New algorithms are developed exclusively using the *composition* operator, which works by taking two existing heuristics H_1 and H_2 and a boolean condition C to create a new heuristic of the form "*if* C *then* H_1 *else* H_2". Ten different conditions were explicated for use with CLASS [7].

The evolved constraint algorithm work of Bain et al. also considers an algorithm to be an expression but in contrast to CLASS its expressions are mathematical: functions and terminals are added and multiplied together to produce a numeric value for each candidate variable. The variable exhibiting the "best" value of this expression (according to a further metric, such as "maximum" or "minimum") is selected as the variable to modify. Genetic programming [9] was selected as an appropriate method of adaptation for such expressions.

The approaches of Bain and Fukunaga are similar in a number of respects. They both can be considered evolutionary algorithms, with the primary difference between the two systems being the genetic operators used to create subsequent generations of algorithms. A discussion of the various merits of these two approaches must consider why evolutionary methods work. The efficacy of

genetic methods are often premised on the existence of *building blocks*, i.e. small clusters of genetic material that imbue an individual with above average performance in the phenotype space [10]. By generating new algorithms, the genetic operators are permitting these building blocks to be tried in new chromosomal contexts. Over a number of generations, repeated selections of above average algorithms cause building blocks to proliferate within the population.

In the standard genetic algorithm with a fixed-length chromosome, evaluating genetic material in new chromosomal contexts necessitates some disruption to an existing chromosome. However, when a chromosome is an expression without a predetermined size or structure, disruption is not necessary and it becomes possible to incorporate all of a parent's genetic material into its children. This is the main difference between the two approaches being considered: the possibility of disruption to a parent's genetic material.

The composition operator used in CLASS does not result in any disruption to 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). Whilst it is advantageous to prevent disruption to and hence the removal of above-average building blocks 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.

One alternative is to use genetic operators, such as crossover and mutation, that can remove some genetic material from candidate expressions. As crossover works by replacing an existing subtree of an expression with a different one, it does not result in such rapid growth in expression size as occurs with composition. Similarly, the problem of redundant information in expressions is also solved by crossover and mutation, as both operators will probabilistically replace some (potentially redundant) parts of an expression with alternate genetic material. Both of these features are at the expense of potential disruption to building blocks that might be responsible for a parent algorithm's above-average performance. The larger the building block, the greater the probability of disruption.

Whether disruption of building blocks will be significant in a particular domain is a difficult question to answer analytically, as it depends on many factors: the fitness landscape of the domain, the composition of the population, and the parameters of the evolutionary algorithm. Without a definitive, analytical answer, the efficacy of genetic operators that preserve the integrity of parent expressions becomes an empirical question.

3 Experimental Study

The purpose of this experimental study is to compare the different genetic operators (crossover and composition) within a common experimental framework. The UBCSAT package was selected as this framework, as it provides a widely recognised experimental environment.

Algorithm expressions are constructed from a set of functions and terminals (constants) that describe properties of the constraint system and the current state of the search. The functions and terminals are based almost entirely on those explicated for use with the CLASS system in [7], which were derived from the material necessary to implement such well-known algorithms as GSAT, WALKSAT and NOVELTY. This representation is strongly-typed to ensure that algorithms constructed from it are functional.

The parameters for the evolutionary procedures are as follows. Both procedures will generate 50 new algorithms each generation (using crossover and composition respectively), with 250 algorithms directly copied from the preceding generation. When duplicating elements, the fittest are selected first, making both experiments "elitist" since the fittest elements can never be lost. This population size was selected to match that used in Fukunaga's original study [7]. Whilst no explicit reason was given for this choice, one plausible explanation concerns the exponential increase in expression size caused by the composition operator. By conveying the majority of the population to the succeeding generation as is, many small algorithms remain to participate in future compositions.

Algorithm expressions are selected to participate in crossover and composition based on their observed fitness value. The raw score of each is taken to be the mean number of flips required to solve 50 randomly generated, phase-transition region, 3-SAT problems: uf100-01 through uf100-050 from the SATLIB benchmark set. Each algorithm expression is permitted 25 tries on each problem, giving a total of 1250 evaluations for each. The cutoff value of 5000 flips is taken as the result of any try for which a solution was not found. The evolutionary procedures were each executed for 50 generations.

Two different fitness equations were considered. As a smaller raw score means a better algorithm, these equations also serve to "standardise" the observed performance so that a smaller score translates into a higher fitness. The first is quite egalitarian, $fitness_i = Best + Worst - score_i$, providing even badly performing algorithms a reasonable chance of being selected. The other equation, $fitness_i = Worst - score_i$, severely punishes the worst performing algorithms so that they are almost never selected. A further option is whether to consider duplicate algorithms eligible (these are truly duplicates or they failed 100% of the time). In both experiments, parent algorithms are selected uniformly at random according to fitness, subject also to whether duplicates are being ignored. If the two algorithms selected would result in a tree exceeding the size bound of 40 nodes, the random choice is repeated up to 1000 times, after which the experiment is terminated.

4 Results and Analysis

The results of experiments conducted with the different selection methods are tabulated in Table 1. These three methods (strict, egalitarian and egalitarian with duplicates) give increasing probability of selection to poorly performing algorithms respectively. In each experiment, the same initial populations were used for crossover as for composition.

Selection Method	Run #	Composition					Crossover				
		Tree Size	Success Rate %	Mean Flips	Mean Time	Rank	Tree Size	Success Rate %	Mean Flips	Mean Time	Rank
Strict	1	37	98.79	1186.09	4.80	1	10	98.68	1186.18	1.99	2
	2	34	98.57	1360.69	4.74	3	37	98.50	1236.28	2.83	4
	3	24	97.47	1494.48	2.77	18	14	98.11	1301.99	1.99	7
Mean		*32*	*98.28*	*1347.09*	*4.10*	*2*	*20*	*98.43*	*1241.48*	*2.26*	*1*
Egalitarian	1	39	98.08	1243.72	3.05	12	17	98.17	1264.89	2.46	6
	2	39	97.82	1358.00	3.63	15	39	98.10	1264.55	3.98	9
	3	38	97.81	1325.15	3.29	16	38	97.84	1394.73	3.27	14
Mean		*39*	*97.90*	*1308.96*	*3.32*	*6*	*31*	*98.04*	*1308.06*	*3.24*	*3*
+Duplicates	1	36	98.09	1247.17	3.78	10	22	98.22	1187.21	2.22	5
	2	36	98.09	1298.19	3.01	11	26	98.11	1456.52	3.79	8
	3	38	97.92	1405.50	3.36	13	22	97.69	1434.38	2.45	17
Mean		*37*	*98.03*	*1316.95*	*3.38*	*4*	*23*	*98.01*	*1359.37*	*2.82*	*5*
Overall Mean		**36**	**98.07**	**1324.33**	**3.60**	**2**	**25**	**98.16**	**1302.97**	**2.77**	**1**

Table 1. Performance of the best evolved algorithms from each experiment, evaluated using a further 1000 runs on each of 100 problems: uf100-01 through uf100-0100.

Performance: Overall, algorithms generated using crossover were marginally more successful by any measure. However, the best individual algorithm was created by the composition operator, solving 98.79% of test runs, slightly more than the best algorithm evolved using crossover, which achieved 98.68% success. The 4 best algorithms were all evolved using the strict fitness function.

The striking difference between these two algorithms is their time performance, with the crossover algorithm averaging only 1.99 seconds to solve each problem, less than half of the 4.80 seconds required by the composition algorithm. This is due to the their tree size: only 10 versus 37 nodes respectively.

Bloat: Bloating is a problem associated with evolutionary methods that don't use a fixed-length representation, occurring when an increasingly complex solution doesn't produce a fitness benefit over that of a simpler solution. One method to control bloating is to incorporate the measure adversely affected by bloat into the fitness function, in this case, run-time. Unfortunately, this transforms the problem into a multi-criteria optimisation problem, introducing additional challenges to balance the use of two disparate fitness measures.

Although not immune from bloat, algorithms generated with crossover exhibited less bloat on average than those generated using composition, having smaller tree sizes (25 versus 36 nodes on average) and a slightly better average performance. This was to be expected by the nature of the composition operator.

Redundancy: In this context, redundancy is considered to be genetic material that does not improve or that worsens performance but has remained present in an algorithm. As composition is unable to remove genetic material from an algorithm, it is plausible that some redundant material is present.

Given the *if..then* nature of the expressions being learned, redundancy may be tested for by replacing an entire *if* conditional with one of its *then* or *else* branches. For example, to test whether the H_1 branch is redundant in the statement "*if* C *then* H_1 *else* H_2", the entire statement would be replaced by H_2.

The above rewriting rule was applied to the best algorithm of both methods. Only the algorithm evolved using composition exhibited redundancy, on 6 of its subtrees. The best of these 6 modified algorithms had a success rate of 98.89%.

5 Conclusion

This work has examined within a common experimental framework two different evolutionary operators for discovering new local search heuristics for solving satisfiability problems. These two operators are the composition operator, a specialised operator for evolving local search heuristics and a standard crossover operator which has been widely used in the genetic programming community. Additionally, three selection methods were examined, each of which gave a different probability of selecting poorer algorithms to participate in recombination.

Although the best performing algorithm was derived using the composition operator, on average algorithms evolved using standard crossover were found to perform better in terms of success, flips and time. Whilst this superior performance was only marginal in the case of success and flips, it was substantial in terms of time, due to the smaller size of the algorithms learned using crossover.

The larger size of the algorithms evolved using composition, without an appreciable fitness benefit, demonstrates the susceptibility of this method to bloat. Crossover was also found to be less susceptible to redundancy, with its best algorithm containing no redundancy whereas a number of redundant subtrees were present in the best algorithm evolved using composition.

In conclusion, giving less consideration to poorly-performing algorithms resulted in the best evolved algorithms. Although composition was able to evolve a number of excellent algorithms for local search, on average, algorithms evolved using a standard crossover recombination operator were found to be superior in terms of success rate, flips and especially time, as well as less susceptible to bloat and redundancy than the corresponding algorithms evolved using composition.

An appendix is available on the author's homepage at http://stuart.multics.org

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. Automatically configuring constraint satisfaction programs: A case study. *Constraints*, 1(1):7–43, 1996.
3. Susan L. Epstein, Eugene C. Freuder, Richard Wallace, Anton Morozov, and Bruce Samuels. The adaptive constraint engine. In *CP '02*, pages 525–540, 2002.
4. Stuart Bain, John Thornton, and Abdul Sattar. Evolving variable ordering heuristics for constrained optimization. In *CP'05.*, page to appear, 2005.
5. Stuart Bain, John Thornton, and Abdul Sattar. Methods of automatic algorithm adaptation. In *PRICAI 2004, LNAI 3157*, pages 144–153, 2004.
6. Dale Schuurmans and Finnegan Southey. Local search characteristics of incomplete SAT procedures. *Artificial Intelligence*, 132(2):121–150, 2001.
7. Alex Fukunaga. Automated discovery of composite SAT variable-selection heuristics. In *Proceedings of AAAI 2002*, pages 641–648, 2002.
8. Alex Fukunaga. Evolving local search heuristics for SAT. In *GECCO-04*, 2004.
9. John Koza. *Genetic Programming: On the programming of computers by means of natural selection.* MIT Press, Cambridge, Massachusetts, 1992.
10. John H. Holland. *Adaptation in natural and artificial systems, 2nd Edition.* MIT Press, Cambridge, Massachusetts, 1992.