Comparison of a Greedy Selection Operator to Tournament Selection and a Hill Climber

Lee Graham, John Borbone, Gary Parker
Department of Computer Science
Connecticut College
New London, Connecticut, USA, 06320
Email: {lgraham3, jborbone, parker} @ conncoll.edu

***Abstract*—A new deterministic greedy genetic algorithm selection operator with very high selection pressure, dubbed the "Jugate Adaptive Method" is examined. Its performance and behavior are compared to those of a canonical genetic algorithm with tournament selection, and a random-restarting next-ascent stochastic hill-climber. All three algorithms are tuned using parameter sweeps to optimize their success rates on five combinatorial optimization problems, tuning each algorithm for each problem independently. Results were negative in that the new method was outperformed in nearly all experiments. Experimental data show the hill climber to be the clear winner in four of five test problems.**

Keywords: greedy selection; hill-climbing; genetic algorithms; combinatorial optimization

I. INTRODUCTION

Evolutionary algorithms and other metaheuristic search techniques attempt to strike a balance between exploitation of candidate solutions already present, and exploration of the search space at large [1-3]. Genetic algorithms (GAs) address the need for exploration through random production of chromosomes during population initialization, by being population-based search techniques, and to some extent by utilizing recombination operators called crossover operators. Exploitation is emphasized through selection mechanisms and mutation operators. Hill-climber (HC) algorithms address the need for exploitation by constantly generating mutants or neighbors of the current candidate solution and adopting superior candidates as they are found. They address the need for exploration by performing restarts – recording the best candidate seen so far, and then jumping to a new randomly-generated candidate solution.

We use the term "greedy" to describe the new selection mechanism (detailed in section III. B.) because of its deterministic and very strong selection pressure. This is not the first time mechanisms of this sort have been employed in genetic algorithms. For instance, the use of *elitism* [7], a common GA heuristic preserving the very best candidate solution(s) between generations, could be considered greedy in the same sense. The use of *truncation selection* [6], is also greedy in our sense. Muehlenbein describes it in the context of the *Breeder Genetic Algorithm* (BGA) in [6] as follows.

"*The BGA uses a selection scheme called truncation selection. The T best individuals are selected and mated randomly until the number of offspring is equal to the size of the population. The offspring generation replaces the parent population. The best individual found so far will remain in the population. Self-mating is prohibited*".

Likewise, the determinism and strictness of the selection mechanism in many hill climbing algorithms [8] is greedy selection in our sense as well.

In this work we present results and analysis of experimental data collected using a new selection method dubbed the Jugate Adaptive Method (JAM) for use in GAs. Earlier work with JAM has shown it to be capable of outperforming both a GA and a HC on some problems [4]. To further investigate its potential, we have attempted here to pit it fairly against both a GA and a HC for a sample of five combinatorial optimization problems: Sorting Networks, the Number Partition Problem, 3-CNF Boolean Satisfiability, Low Autocorrelation Binary Sequences, and a two-dimensional Ising Spin Glass problem. Each algorithm is tuned independently for each problem using parameter sweeps to tap its full potential. To keep comparisons between algorithms fair, the total number of fitness evaluations per run is held constant for a given problem, regardless of the algorithm.

The success rates – the fraction of independent runs that attain optimal fitness – are used as a basis for comparison between algorithms. We also consider the sensitivity of each algorithm-problem pair to the mutation rate.

Section II, below, describes each of the five sample problems used in these experiments. Section III explains the three algorithm implementations: a GA, JAM, and a HC. Section IV presents the experimental results, and section V contains observations and concluding remarks.

II. OPTIMIZATION PROBLEMS

The following five sample problems were used in our experiments. They are a sorting network problem [11, 12], a number partitioning problem [10], a Boolean satisfiability problem [9], a low autocorrelation binary sequence problem [5], and an Ising spin glass problem [13]. In each case, the GA, JAM, and HC implementations were tuned on the individual

problems using parameter sweeps. The subsections below describe these problems.

A. Sorting Network Problem (SN)

A sorting network [11, 12] is an oblivious sorting algorithm represented by a combinatorial object. It is oblivious in that the sequence of element comparisons performed during its operation is fixed; specific comparison operations chosen are not influenced by the outcome of earlier comparisons. Sorting networks, because they are oblivious in this way, lend themselves well to hardware implementation. The combinatorial object representing such an algorithm is merely a sequence of comparators. A comparator is simply an unordered pair of element indices. To sort an array of N inputs, the comparators are processed in order. Each comparator examines the values at its two indices and swaps them iff they are out of order.

The particular sorting network problem instance used here is the 7-input problem (sorting an array of seven values) using exactly 16 comparators, which is known to be the minimum number necessary to solve the 7-input problem. Candidate solutions are represented by a sequence of 32 indices, each in the range zero to six, inclusive. Taken two at a time, they completely specify the 16 comparators that make up the sorting network. Candidate networks are evaluated by attempting to sort all possible arrays in which the elements are either zero or one. There are $2^7 = 128$ such input arrays, and by the so-called zero-one principle, successfully sorting all of these input is sufficient to ensure the network works correctly in the general case. This is a maximization problem where fitness is the total number of input lists successfully sorted. An independent run of any of the three algorithms in question counts as a success iff its best-of-run fitness is exactly 128.

B. Number Partitioning Problem (NPP)

NPP is a special case of the more general NP-complete problem called Subset Sum. The problem is, given a list of numbers, can a subset be found for which the sum of its elements is exactly half of the sum of the original list [10]. Put another way, the problem is to determine whether the numbers can be divided into two subsets of equal sum.

The problem instance used here is a list of 200 integers that is known to be solvable. Candidate solutions are represented as bit strings of length 200 where bit i, $0 \leq i < 200$ is used to assign element i to one of the two subsets. It is a minimization problem with fitness being the absolute value of the difference between the two subsets' sums. Each run of one of the algorithms is counted as a success if it finds a solution with fitness value zero.

C. Boolean Satisfiability (3SAT)

3SAT is a 3-CNF (conjunctive normal form) Boolean satisfiability problem. This is one of the most well-known NP-complete problems [9] in which one must determine whether a given Boolean formula can be satisfied. In 3SAT, the formula takes the form of a conjunction of C clauses where each clause is a disjunction of three literals (a literal is a Boolean variable or its negation).

The problem instance used here is a formula consisting of 425 clauses and 100 distinct Boolean variables. The instance is known to admit to a solution. Candidate solutions are represented as bit strings of length 100 where bit i, $0 \leq i < 100$, dictates the truth value assigned to variable i in the formula. This is a maximization problem where fitness is a floating point value: the number of clauses satisfied plus (L / 425) where L is the total number of true literals in the formula. Any run attaining fitness greater than or equal to 425 is considered successful.

D. Low Autocorrelation Binary Sequence (LABS)

The LABS problem [5] is known to be particularly difficult to handle with local search algorithms. It involves minimizing the *energy function* of a binary sequence of a given length. This function is calculated as

$$E(S) = \sum_{k=1}^{n-1} \left[\sum_{i=1}^{n-k} s_i s_{i+k} \right]^2 , \tag{1}$$

where S is the binary sequence $\{s_1, s_2, .., s_n\}$.

The problem instance used here is LABS with length-22 binary sequences. This problem instance has a known minimum E(S) value of 39. Any run reaching that value is counted as a successful run.

E. Ising Spin Glass Problem (Ising)

The Ising Spin Glass problem [13] comes from the field of statistical mechanics and models the interactions of particles in a lattice where each particle has one of two possible *spin* values and interacts with its closest neighbors. The problem can be represented by a graph where each node corresponds to a particle and each edge corresponds to an interaction between the two particles it connects. Interactions come in two types, ferromagnetic and antiferromagnetic, represented by a +1 or a -1 associated with the edge. The goal of the problem is to set the spin values for each particle (either +1 or -1) so as to minimize the energy function

$$H = -\sum_{e \in E} e_i s(e) , \tag{2}$$

where E is the set of edges, e_i is the interaction value associated with edge e, and s(e) is the product of the two spin values at either end of edge e.

The problem instance used here is a two-dimensional toroidal lattice of size 10x10 in which each node is attached by an edge to the adjacent neighbor on its right and to the neighbor directly below it. The problem instance in use has a known minimum energy value of -138. The energy value is used as the fitness value, and candidate solutions are represented as binary strings of length 100 with each bit determining the spin value for a unique node in the graph. Any run finishing with best fitness equal to -138 is a successful run.

III. Three Algorithms

The three algorithms being compared here are a GA, an implementation of JAM, and a next-ascent hill-climber. The behavior and parameters of each algorithm are described in the subsections below.

A. Genetic Algorithm(GA)

The GA implementation is a relatively typical generational GA with tournament selection and single-individual elitism (the single best chromosome is copied unchanged from one generation to the next). When a new offspring is to be produced, either one or two tournaments are held to choose parents, depending on the crossover rate. Mutation is applied after crossover (in the case of two parents) or after cloning (in the case of one parent) and the mutation rate is a per-gene probability of choosing a new (and different) value uniformly at random from within the legal range for the gene. The GA terminates a run whenever the total allotted fitness evaluations have been used up, even if this occurs in the middle of a generation. The best fitness value encountered since the run began is recorded for each independent run performed.

The only aspect of the GA that is somewhat atypical is that the initial randomly-generated population has size M+F where M is the population size and F is a parameter called *filter evaluations*. Once the M+F randomly-generated individuals have been evaluated, the worst F individuals are discarded and the GA proceeds in the normal fashion with a fixed population size of M.

The parameters of the GA are 1) total allotted evaluation count, 2) number of filter evaluations, 3) population size, 4) crossover operator (*uniform*, *single-point*, or *two-point*), 5) crossover rate, 6) mutation rate, and 7) tournament size.

B. Jugate Adaptive Method (JAM)

The JAM algorithm is a generational GA implementing a deterministic greedy selection method. Population initialization follows the same scheme as the GA described above, including the use of *filter evaluations*. Selection, however, proceeds as follows. At the start of a new generation, two individuals – always the fittest and second fittest individuals – are found and the fittest individual is copied unchanged into the next generation. The remaining M–1 offspring (where M is population size) are produced by either a) mutation applied to a clone of the fittest individual, or b) mutation applied to the result of crossover between the fittest and second fittest individuals. The crossover rate parameter determines the probability of choosing b) over a), and the mutation rate parameter determines the severity of the mutation operator. In any given domain and implementation, the meaning of the mutation rate will depend on the nature of the mutation operator itself. For example, it may be the probability associated with bit-toggling in a binary string chromosome. Most likely because of the severe drop in diversity between generations, caused by this form of selection, the JAM algorithm, as will be seen in the results section below, generally operates best at higher mutation rates than the GA.

The parameters of this JAM implementation are 1) total allotted evaluation count (the halting condition is thus exhaustion of this limited number of calls to fitness evaluation), 2) number of filter evaluations, 3) choice of crossover operator (*uniform*, *single-point*, or *two-point*), 4) crossover rate (probability of using crossover to create a given offspring), and 5) mutation rate (severity of the mutation operator).

C. Hill Climber (HC)

The HC implementation uses a next-ascent strategy and produces neighbors using the same mutation operator as the GA and JAM do. This is a simple mutation operator that visits each allele and changes its value with probability equal to mutation rate. Unlike the GA and JAM, the HC applies the mutation operator repeatedly until at least one gene is modified. The HC will restart (i.e. return to a randomly-generated candidate solution to begin a new climb) whenever either of two criteria are met: a) the total number of neutral steps (replacements of the current candidate solution with one of equal fitness), since the last ascent or restart, exceeds a fixed threshold, or b) the total number of strictly less fit offspring produced, since the last ascent or restart, exceeds a fixed threshold. The HC will continue climbing, wandering across fitness plateaus, and restarting, until the overall total allotted fitness evaluations have been consumed.

The parameters of this HC implementation are 1) total allotted evaluation count, 2) plateau threshold (number of neutral steps before restart is triggered), 3) neutral step probability (the probability that offspring with equal fitness will be adopted), 4) mutation rate, and 5) dead end threshold (number of strictly less fit offspring before restart is triggered).

IV. Results

With the exception of total allotted evaluation count, which was fixed for any given problem and for all algorithms when applied to that problem, each of the parameters for each of the three algorithms on each of the five sample problems was tuned using parameter sweeps. To tune algorithm A to problem P, each of A's parameters was taken in turn, and numerous independent runs were averaged for a wide sample of values within the parameter's range. The value yielding the best mean success rate (successful runs divided by total runs) became fixed before moving on to sweep the next parameter.

Once each algorithm was tuned separately for each problem, a final parameter sweep for mutation rate was conducted in order to see, for each algorithm-problem pair, how sensitive the tuned systems are to mutation rate.

TABLE I. Success Rates For Each Algorithm-Problem Pair

Problem	Success Rates (with 95% confidence intervals)		
	GA	***JAM***	***HC***
SN	2.2% ± 0.52%	1.3% ± 0.41%	**6.7% ± 0.49%**
NPP	36% ± 1.7%	33% ± 1.7%	37% ± 1.3%
3SAT	19% ± 1.4%	7.6% ± 0.95%	**24% ± 1.2%**
LABS	18% ± 1.4%	38% ± 1.7%	**69% ± 1.7%**
Ising	0.46% ± 0.19%	0.06% ± 0.06%	**21% ± 1.1%**

Table 1 shows the success rates for each of the algorithm-problem pairs, including 95% confidence intervals. Success rates in underlined bold font signify the winning algorithm for each problem. For NPP the GA and the HC are effectively tied for first place. It is immediately clear upon inspection that the HC algorithm has outperformed both the canonical GA and the JAM implementations in four out of the five problems, usually by a substantial amount, and is tied for first place on the remaining problem.

The two-individual inter-generational bottleneck in the JAM algorithm would seem to place it somewhere between a GA and a HC, but in terms of performance on these particular sample problems, this is not borne out by the data. In three out of five cases, the JAM algorithm performs much more poorly than the other two algorithms. Only in the case of the LABS problem does the ranking place JAM between GA and HC.

Figure 1, below, shows these data in graphical form.

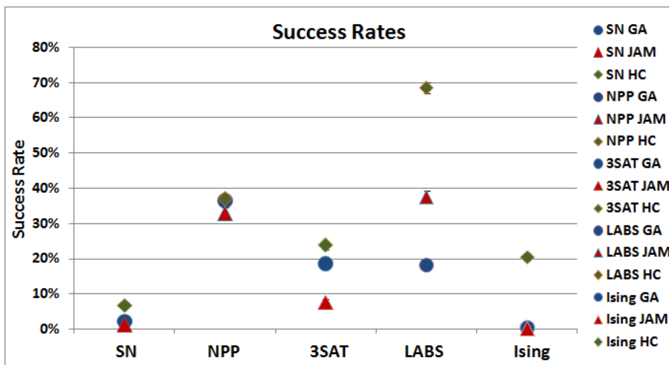

Figure 1. Mean best-of-run success rates for each algorithm-problem pair, including 95% confidence intervals.

Figures 2, 3, 4, 5, and 6, below, show the results from the final mutation rate sweep on the tuned implementations of all three algorithms on the Ising Spin Glass problem, the Sorting Network problem, the LABS problem, the NPP problem, and the 3-CNF Boolean satisfiability problem, respectively.

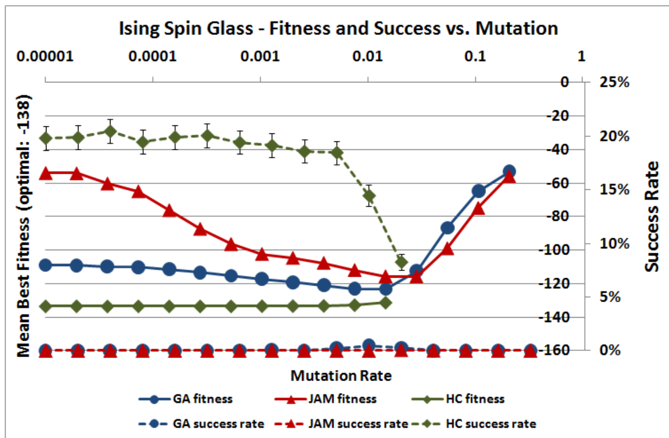

Figure 2. Mean best-of-run fitness and success rate vs. mutation rate for the Ising Spin Glass problem.

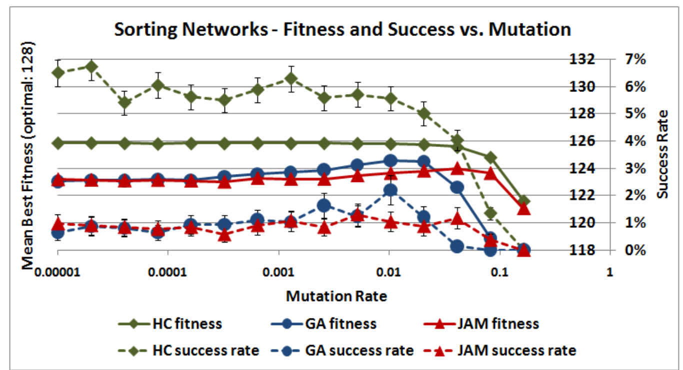

Figure 3. Mean best-of-run fitness and success rate vs. mutation rate for the Sorting Network problem.

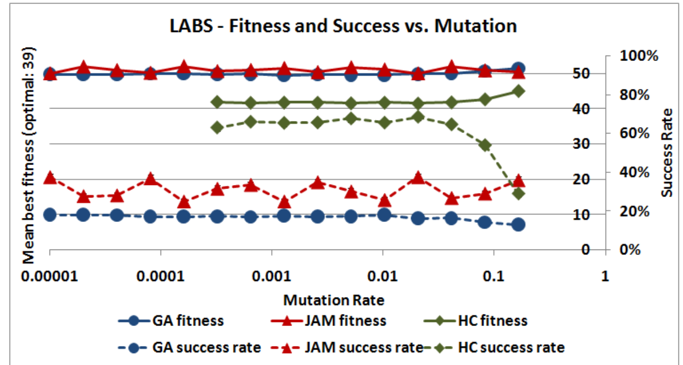

Figure 4. Mean best-of-run fitness and success rate vs. mutation rate for the LABS problem.

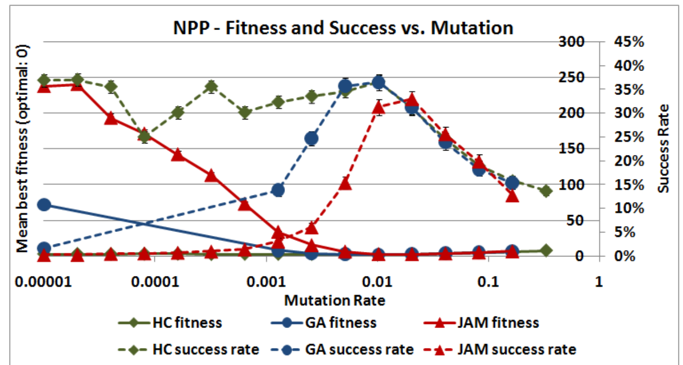

Figure 5. Mean best-of-run fitness and success rate vs. mutation rate for the NPP problem.

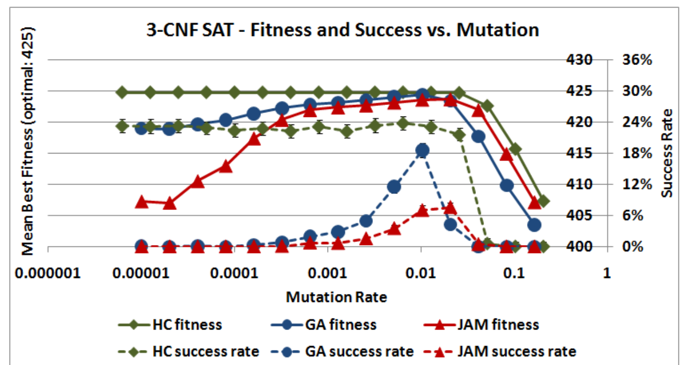

Figure 6. Mean best-of-run fitness and success rate vs. mutation rate for the 3-CNF Boolean Satisfiability problem.

V. Observations and Conclusions

Although the results based on the five sample problems used here do not involve JAM outperforming GA and HC, there are a few observations to be made based on the data presented. Firstly, the mutation sweeps done on the tuned algorithm-problem pairs fairly consistently show a reverse S-shape success rate graph for the HC algorithm where the other algorithms' success rate graphs tend to have a central peak that slopes downward to either side. This may be due to the different approach to mutation taken by the HC algorithm. Where the GA and JAM algorithms simply call the mutation operator, the HC algorithm does so inside a while-loop in order to guarantee that at least one allele has been modified. It may be worthwhile to add this feature to JAM (and perhaps to the GA) in order to see whether it a) alters the shape of the success rate graphs, and/or b) improves overall performance.

Second, as was expected, the JAM algorithm exhibits its peak performance at higher mutation rates than does the GA at its peak. In some cases, the optimal mutation rate is approximately twice that of the GA. This is very likely due to the need in JAM to balance the severe diversity loss associated with the two-individual bottleneck.

Third, although the effect is less prominent once the algorithms have been tuned, the JAM algorithm appears to be less sensitive to mutation rate than the GA. This became apparent during the tuning process when mutation rate parameter sweeps were being conducted before the other parameters had been tuned.

Further experimentation with the JAM algorithm will be pursued in future. Although it was beaten by the HC in all five sample domains, it outperformed the GA in at least one – the LABS problem. This problem is known to be particularly difficult for search algorithms of this nature [5], so that result is promising. Some lessons might be learned based on these experiments that can guide further augmentation and modification of the JAM algorithm (perhaps, for example, it may benefit from a HC-like restart mechanism, as it is already arguably a hybridization of GA and HC search strategies).

References

[1] D. E. Goldberg, *Genetic Algorithms in Search Optimization and Machine Learning*. Addison Wesley, 1989.

[2] D. B. Fogel, Evolutionary Computation: Toward a New Philosophy of Machine Intelligence. New York: IEEE Press, 2000.

[3] T. Bäck, *Evolutionary Algorithms in Theory and Practice*. Oxford University Press, 1996.

[4] J. Borbone, L. Graham, G. Parker, and C. Chung, *Testing a JAM (Jugate Adaptive Method) Selection Method Against Other Standard Algorithms using Varying Test Environments and Fitness Functions*. In press.

[5] I. Dotu, and P. Van Hentenryck, *A Note on Low Autocorrelation Binary Sequences*. F. Benhamou (Ed.): CP 2006, LNCS 4204, pp. 685-689, Springer Verlag Berlin Heidelberg, 2006.

[6] Muehlenbein, H., and Schlierkamp-Voosen, D. (1993) "Predictive Models for Breeder Genetic Algorithm" in Evolutionary Computation, vol 1, p. 25-49.

[7] De Jong, K. A. (1975) "An Analysis of the Behavior of a Class of Genetic Adaptive Systems", PhD thesis, University of Michigan, Ann Arbor.

[8] Michalewicz, Z., and Fogel, D. B. (1998) "How to Solve It: Modern Heuristics", Springer. Section 2.6, p. 43-44.

[9] Cook, S. (1971). "The complexity of theorem proving procedures", in Proceedings of the Third Annual ACM Symposium on Theory of Computing. pp. 151–158.

[10] Du, D.-Z., and Ko, K.-I. (2000) "Theory of Computational Complexity", Wiley. Chapter 2, p. 57. {Note: *"PARTITION" is the problem herein referred to as "NPP"}*

[11] Knuth, D. (1998) "The Art of Computer Programming, Volume 3: Sorting and Searching" (2nd edition). Addison Wesley.

[12] Juillé, H. (1995) "Evolution of Non-deterministic Incremental Algorithms as a New Approach for Search in State Spaces". In Proceedings of ICGA-95. Morgan Kaufmann, pp. 351-358.

[13] Nishimori, Hidetoshi (2001). "Statistical Physics of Spin Glasses and Information Processing: An Introduction". Oxford: Oxford University Press. pp. 243.