An Empirical Comparison of Seven Iterative and Evolutionary Function Optimization Heuristics

Shumeet Baluja
September 1, 1995
CMU-CS-95-193

School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

baluja@cs.cmu.edu

Abstract

This report is a repository for the results obtained from a large scale empirical comparison of seven iterative and evolution-based optimization heuristics. Twenty-seven static optimization problems, spanning six sets of problem classes which are commonly explored in genetic algorithm literature, are examined. The problem sets include job-shop scheduling, traveling salesman, knapsack, binpacking, neural network weight optimization, and standard numerical optimization. The search spaces in these problems range from 2^{368} to 2^{2040}. The results indicate that using genetic algorithms for the optimization of static functions does not yield a benefit, in terms of the final answer obtained, over simpler optimization heuristics. Descriptions of the algorithms tested and the encodings of the problems are described in detail for reproducibility.

This work was started while the author was supported by a National Science Foundation Graduate Fellowship. He is currently supported by a graduate student fellowship from the National Aeronautics and Space Administration (NASA), administered by the Lyndon B. Johnson Space Center. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the National Science Foundation, NASA, or the U.S. Government.

Keywords

Heuristic Static Function Optimization, Evolutionary Algorithms, Genetic Algorithms, Hillclimbing, Population-Based Incremental Learning

1. INTRODUCTION

Genetic algorithms (GAs) and other evolutionary procedures are commonly used for static function optimization. Although there has been growing evidence that methods such as GAs are, in general, not well suited in this domain [De Jong, 1992], a large amount of research has been devoted to improving their effectiveness for function optimization. Hybrid mechanisms, ranging from alternate evolutionary methods to specialized operators and representations which can intelligently use problem specific information, have achieved good results in many specific applications. Nonetheless, relatively few of these techniques work well across a wide range of problems.

The aim of this paper is to compare two standard genetic algorithms with simpler methods of optimization: multiple-restart stochastic hillclimbing (MRSH) and population-based incremental learning (PBIL). Previous comparisons between forms of MRSH and GAs can be found in [Ackley, 1994], [Juels & Wattenberg, 1994], [Forrest & Mitchell, 1992], [Mitchell & Holland, 1994], and [Davis, 1991], to name a few. A comparison between GAs and PBIL has been made in [Baluja, 1994][Baluja & Caruana, 1995]. This paper provides a large scale empirical comparison of these algorithms on problems commonly found in GA literature. Three variants of MRSH, two variants of PBIL, and two GAs are compared.

1.1 The Aims of this Paper

This study aims at answering only one question: "How effective are standard GAs for optimizing static functions, given a set number of function evaluations, in comparison to other, simpler, algorithms?" This paper presents results on many large problems; the size and quantity of the problems makes it hard to give in-depth analysis of the results beyond the algorithms' relative performances. A more in-depth analysis of PBIL in comparison to standard GAs on a problem which was specifically designed to be easy for the genetic algorithm (and easier to analyze than the problems explored here) is provided in [Baluja & Caruana, 1995]. This paper does not attempt to address the problem of whether the classes of problems investigated are suited for evolutionary or iterative function optimization. The focus of this paper is on comparing seven static function optimization methods on problems which are representative of problems commonly used as benchmarks in GA literature. No problem specific features have been added to any of the algorithms; all of the mechanisms used in the algorithms are "standard", and have been explored and described in the applicable literature. The inclusion of problem specific mechanisms or more sophisticated features has the potential to improve the performance of all the algorithms.

There are two major concerns with performing a purely empirical comparison of these algorithms. The first is that each of these algorithms is defined by control parameters, and it is prohibitively expensive, in practice, to thoroughly explore the space of the parameters while providing breadth in the types and sizes of problems attempted. The GA parameters used here were chosen to work well on many of the problems, but are not biased to any particular single problem. The parameters for the other algorithms were chosen in the same manner. In addition, GAs were selected to perform well on the task of optimization; they use mechanisms, such as elitist selection and scaling of fitness values, which are often used for the optimization of static functions [De Jong, 1992]. *One of the goals of this study is to use the algorithms with as little problem-specific knowledge as possible*. The only problem-specific knowledge used in these algorithms is the number of bits in the solution encoding for each of the problems.

The second concern is that there are many criteria by which the effectiveness of each algorithm can be measured. As mentioned before, there has recently been some controversy in the GA community as to whether GAs should be used for static function optimization. One of the reasons for this controversy is that GAs “attempt to maximize the cumulative payoff of a sequence of trials” [De Jong, 1992] rather than attempt to find the single best optimum. Therefore, using the “best answer found” criteria may not be the best way to measure the GA’s abilities. Nonetheless, a considerable amount of effort has been devoted to making the GAs better in function optimization. “Better” has usually been measured in terms of the best solution found in a given number of trials. The common forms of measurement for function optimization are on-line and off-line performance. On-line performance measures the average of all function evaluations up to and including the current evaluations. Off-line performance is a running average of the best performance values to a particular time. Other measurements include the best solution found in the final generation and the best solution found in any generation through the search. Although all these measures reveal different insights into the search algorithm’s ability, the measure we are interested in this study is the best solution ever found through the search. The issues of cumulative payoff, on-line and off-line performance are not addressed here. The effectiveness of each algorithm is based solely upon the best answer it can find in the given number of trials.

It is important to understand the scope of these results. All of the empirical comparisons are based upon static function optimization problems. The performance of each method is judged solely by the best solution found during the run, given a pre-specified number of total evaluations. Therefore, the following classes of problems are ***not*** considered here, and should be explored in the future:

- Noise in the evaluation function [Grefenstette & Fitzpatrick, 1988].
- A changing, or time-varying, evaluation function (over the period of a single run) [Cobb, 1993].
- Problems in which queries have an associated cost, which must also be minimized [Cohn, 1994].
- Problems in which multiple “solution vectors” must interact [Langton, 1994].
- Problems in which cumulative payoff is to be optimized [Holland, 1975], [Golberg, 1989].
- Problems which use variable-length encodings, or encodings with change over time [Koza, 1992].

Although the above domains are not addressed here, the domain which is concentrated upon covers a wide variety of problems. A large portion of GA research has been devoted to the types of problems analyzed in this paper. The field of Operations Research is another source of many similar problems.

The next section describes the simplest algorithm tested, multiple-restart stochastic hillclimbing. This section is followed by descriptions of genetic algorithms, in section 3, and population-based incremental learning in section 4. In section 5, the problems attempted and the results obtained are described together. Section 6 summarizes the empirical results. Section 7 concludes the report and suggests some areas for future studies.

2. MULTIPLE-RESTART STOCHASTIC HILLCLIMBING

Multiple-restart stochastic hillclimbing (MRSH) is a method of iterative optimization of static functions. It is the simplest of the optimization procedures explored in this paper. [Wattenberg and Juels, 1994] have compared one version of stochastic hillclimbing with GAs on several problems commonly used for gauging genetic algorithms and genetic programming, and have achieved very promising results. The basic stochastic hillclimbing algorithm is shown in Figure 1.

```
V ← randomly generate solution vector
Best ← evaluate (V)

loop # ITERATIONS
        N ← Flip_Random_Bit (V)
        if (evaluate (N) > Best)
                Best ← evaluate(N)
                V ← N
```
Flip_Random_Bit is a function which returns a solution string with only one bit changed from its input solution string.

Figure 1: The stochastic hillclimbing algorithm for binary solution vectors. In the full algorithm, the best vector along with its evaluation would be saved. In practice the algorithm could be restarted in random locations many times - and the best solution ***ever*** found returned.

Three variants of this algorithm are explored in this paper. The first variant, (MRSH-1) maintains a list of the position of the bit flips which were attempted without improvement. These bit flips are not attempted again until a better solution is found. When a better solution is found, the list is emptied. If the list becomes as large as the solution encoding, then no single bit flip can improve the solution. In this case, MRSH-1 is restarted at a random location with an empty list.

The second and third variants of stochastic hillclimbing, (MRSH-2 & MRSH-3), allow moves to regions of *higher and equal* evaluation. This is different than MRSH-1, which only allows moves to regions of higher evaluation. MRSH-2 & 3 differ from each other in the number of evaluations allowed before restarting search in a random location. In MRSH-2, the number of evaluations is dependent upon the length of the encoded solution. MRSH-2 allows 10*(length of solution) evaluations *without improvement* before search is restarted. When a solution with a higher evaluation is found, the count is reset. MRSH-3 enforces a much stricter policy of restart; after the total number of iterations is specified, restart is forced 5 times during search, at equally spaced intervals.

3. GENETIC ALGORITHMS

Genetic algorithms (GAs) are biologically motivated adaptive systems which are based upon the principles of natural selection and genetic recombination. A GA combines the principles of survival of the fittest with a randomized information exchange. It has the ability to recognize trends toward optimal solutions, and to exploit such information by guiding the search toward them.

In the standard GA, candidate solutions are encoded as fixed length vectors. The initial group of potential solutions is chosen randomly. These candidate solutions, called "chromosomes," are allowed to evolve over a number of generations. At each generation, the fitness of each chromosome is calculated; this is a measure of how well the chromosome optimizes the objective function. The subsequent generation is created through a process of selection, recombination, and mutation. The chromosomes are probabilistically selected for recombination based upon their fitness. General recombination (crossover) operators merge the information contained within pairs of selected "parents" by placing random subsets of the information from both parents into their respective positions in a member of the subsequent generation. Although the chromosomes with high fitness values have a higher probability of selection for recombination than those with low fitness values, they are not guaranteed to appear in the next generation. Due to the random factors involved in producing "children" chromosomes, the children may, or may not, have higher fitness values than their parents. Nevertheless, because of the selective pressure applied through a number of generations, the overall trend is towards higher fitness chromosomes. Mutations are used to help preserve diversity in the population. Mutations introduce random changes into the chromosomes. A good overview of GAs can be found in [Goldberg, 1989] [De Jong, 1975].

Two variants of the traditional genetic algorithm are tested in this study. The first, SGA, has the following parameters: Two-Point crossover, with a Crossover Rate of 100%, Mutation Rate: 0.001, Population Size: 100, Elitist selection (the best chromosome in generation N replaces the worst chromosome in generation N+1). The second GA used, termed GA-Scale, uses the same parameters, with the following exceptions: Uniform crossover with a crossover rate of 80%, and the fitness of the worst member in a generation is subtracted from the fitnesses of each member of the generation before the probabilities of selection are determined. Both GAs are generational, and both employ the elitist selection mechanism described above.

4. POPULATION-BASED INCREMENTAL LEARNING

Population-based incremental learning (PBIL) is a combination of evolutionary optimization and hill-climbing [Baluja, 1994]. The object of the algorithm is to create a real valued probability vector which, when sampled, reveals high quality solution vectors with high probability. For example, if a good solution to a problem can be encoded as a string of alternating 0's and 1's, a suitable final probability vector would be 0.01, 0.99, 0.01, 0.99, etc.

Initially, the values of the probability vector are set to 0.5. Sampling from this vector yields random solution vectors because the probability of generating a 1 or 0 is equal. As search progresses, the values in the probability vector gradually shift to represent high evaluation solution vectors. This is accomplished as follows: A number of solution vectors are generated based upon the probabilities specified in the probability vector. The probability vector is pushed towards the generated solution vector(s) with the highest evaluation. The distance the probability vector is pushed depends upon the learning rate parameter. After the probability vector is updated, a new set of solution vectors is produced by sampling from the updated probability vector, and the cycle is continued. As the search progresses, entries in the probability vector move away from their initial settings of 0.5 towards either 0.0 or 1.0. The probability vector can be viewed as a prototype vector for generating solution vectors which have high evaluations with respect to the available knowledge of the search space.

This algorithm is an extension of the Equilibrium Genetic Algorithm developed in conjunction with [Juels, 1993, 1994]. Another algorithm related to EGA/PBIL is Bit-Based Simulated Crossover (BSC) [Syswerda, 1992][Eshelman & Schaffer, 1993]. BSC regenerates the probability vector at each generation; it also uses selection probabilities (as do standard GAs) to generate the probability vector. In contrast, PBIL does not regenerate the probability vector at each generation, rather, the probability vector is updated through the search procedure. Additionally, PBIL does not use selection probabilities. Instead, it updates the probability vector using a few (in these experiments 1) of the best performing individuals.

The manner in which the updates to the probability vector occur is similar to the weight update rule in supervised competitive learning networks, or the update rules used in Learning Vector Quantization (LVQ) [Hertz, Krogh & Palmer, 1993]. Many of the heuristics used to make learning more effective in supervised competitive learning networks (or LVQ), or to increase the speed of learning, can be used with the PBIL algorithm. This relationship is discussed in greater detail in [Baluja, 1994].

4.1 PBIL's Relation to Genetic Algorithms

One key feature of the *early* portions of genetic optimization is the parallelism in the search; many diverse points are represented in the population of early generations. As the search progresses, the population of the GA tends to converge around a good solution vector in the function space (the respective bit positions in the majority of the solution strings converge to the same value). PBIL attempts to create a probability vector that is a prototype for high evaluation vectors for the function space being explored. As search progresses in PBIL, the values in the probability vector move away from 0.5, towards either 0.0 or 1.0. Analogously to genetic search, PBIL converges from initial diversity to a single point where the probabilities are close to either 0.0 or 1.0. At this point, there is a high degree of similarity in the vectors generated.

Because PBIL uses a single probability vector, it may seem to have less expressive power than a GA using a full population that can represent a large number of points simultaneously. For example, in Figure 2, the vector representations for populations #1 and #2 are the same although the members of the two populations are quite different. This appears to be a fundamental limitation of PBIL; a GA would not treat these two populations the same. A traditional single population GA, however, would not be able to *maintain* either of these populations. Because of sampling errors, the population will converge to one point; it will not be able to maintain multiple dissimilar points. This phenomenon is summarized below:

> "... the theorem [Fundamental Theorem of Genetic Algorithms [Goldberg, 1989]], assumes an infinitely large population size. In a finite size population, even when there is no selective advantage for either of two competing alternatives... the population will converge to one alternative or the other in finite time (De Jong, 1975; [Goldberg & Segrest, 1987]). This problem of finite populations is so important that geneticists have given it a special name, genetic drift. Stochastic errors tend to accumulate, ultimately causing the population to converge to one alternative or another" [Goldberg & Richardson, 1987].

Similarly, PBIL will converge to a probability vector that represents one of the two solutions in each of the populations in Figure 2; the probability vector can only represent one of the dissimilar points.

In addition to moving the prototype vector towards the highest evaluation vector, the prototype vector can also be moved away from the lowest evaluation vector generated in each generation. However, as the prototype vector becomes fixed towards either 0.0 or 1.0 for each bit position, the hamming distance between the best and worst generated vectors will diminish. If the hamming distance between the best and worst vector is small, moving away from the worst vector is counter-productive, because it also moves away from the best vector in many of the bit positions. Instead, the probability vector can be moved away from the values in the worst vector which differ from those in the respective positions of the best vector. The full algorithm is shown in Figure 3.

In this study, two variants of the algorithm shown in Figure 3 are used. The first, PBIL, uses the following parameters: Mutation Probability: 0.02, Mutation Shift: 0.05, Learning Rate: 0.1, and Negative Learning

Population #1	Population #2
0 0 1 1	1 0 1 0
1 1 0 0	0 1 0 1
1 1 0 0	1 0 1 0
0 0 1 1	0 1 0 1
Representation	**Representation**
0.5, 0.5, 0.5, 0.5	0.5 0.5 0.5 0.5

Figure 2: The probability representation of 2 small populations of 4-bit solution vectors; population size is 4. Notice that the representations for both populations are the same, although the solution vectors represented are entirely different.

Rate: 0.075. The second algorithm, the PBIL/EGA algorithm, uses the same parameters with the Negative Learning Rate set to 0.0.

5. AN EMPIRICAL COMPARISON

In this section, the algorithms described previously are applied to six classes of problems: Traveling Salesman, jobshop scheduling, knapsack, bin packing, neural network weight optimization, and numerical function optimization. The results obtained in this study should *not* be considered to be state-of-the-art. The problem encodings were chosen to be easily reproducible, and to allow easy and fair comparison with other studies. Alternate encodings may yield superior results. In addition, no problem-specific information was used for any of the algorithms. In the cases in which problem-specific information is available, it may be able to help all of the search algorithms presented in this study.

In the problems presented in this paper, all of the variables were encoded either with Gray-code or standard base-2 representation, as indicated with the problem. The variables were represented in non-overlapping, contiguous positions within the chromosome (solution encoding). The results reported are the best evaluations found through the search of each algorithm, averaged over at least 20 independent runs per algorithm per problem. In the problems in which random values are assigned to problem attributes (such as the location of cities in the Traveling Salesman Problems or sizes of elements in the bin packing and knapsack problems), the values are consistent across all algorithms attempted and across all 20 trials for each algorithm.

All algorithms were allowed an equal number of evaluations per run (200,000). In each run, the GA and PBIL algorithms both were allowed 2000 generations, with 100 function evaluations per generation. In each run, the MRSH algorithms were restarted in random locations as many times as needed until 200,000 evaluations were performed. The best answer ever found in the 200,000 evaluations was returned as the best answer found in the run. The final results for the problems are given in tables following the description of the problems. The best results are highlighted.

5.1 Traveling Salesman Problems (TSP)

The TSP problem is probably the most famous of the NP-complete problems. Given **N** cities, the object is to find a minimum length tour which visits each city exactly once. The encoding used in this study requires a bit string of size $\mathbf{N}\log_2\mathbf{N}$ bits. Each city is assigned a substring of length $\log_2\mathbf{N}$ which is interpreted as an integer. The city with the lowest integer value comes first in the tour, the city with the second

```
****** Initialize Probability Vector *****
for i :=1 to LENGTH do P[i] = 0.5;

while (NOT termination condition)
        ***** Generate Samples *****
        for i :=1 to SAMPLES do
                sample_vectors[i] := generate_sample_vector_according_to_probabilities (P);
                evaluations[i] :=Evaluate_Solution (sample[i]);

        best_vector := find_vector_with_best_evaluation (sample_vectors, evaluations);
        worst_vector := find_vector_with_worst_evaluation (sample_vectors, evaluations);

        ***** Update Probability Vector towards best solution *****
        for i :=1 to LENGTH do
                P[i] := P[i] * (1.0 - LR) + best_vector[i] * (LR);


        ***** Update Probability Away from Worst Solution *****
        for i :=1 to LENGTH do
                if (best_vector[i] ≠ worst_vector[i]) then
                        P[i] := P[i] * (1.0 - NEGATIVE_LR) + best_vector[i] * (NEGATIVE_LR);


        ***** Mutate Probability Vector *****
        for i :=1 to LENGTH do
                if (random (0,1) < MUT_PROBABILITY) then
                        if (random (0,1) > 0.5) then mutate_direction := 1
                        else mutate_direction := 0;
                        P[i] := P[i] * (1.0 - MUT_SHIFT) + mutate_direction * (MUT_SHIFT);
```

USER DEFINED CONSTANTS (Values Used in this Study):

SAMPLES: the number of vectors generated before update of the probability vector (100).

LR: the learning rate, how fast to exploit the search performed (0.1).

NEGATIVE_LR: the negative learning rate, how much to learn from negative examples (PBIL = 0.075, EGA = 0.0).

LENGTH: the number of bits in a generated vector (problem specific).

MUT_PROBABILITY: the probability for a mutation occurring in each position (0.02).

MUT_SHIFT: the amount a mutation alters the value in the bit position (0.05).

Figure 3: The PBIL/EGA algorithm for a binary alphabet.

lowest comes second, etc. In the case of ties, the city whose substring comes first in the bit string comes first in the tour. This encoding was taken from [Syswerda, 1992]. To minimize the tour length, the evaluation used is 1.0/**Tour_Length**. Four problems were attempted: the first contained 128 cities, the second contained 200 cities, and the third and fourth contained 255 cities. The integer encoding of the fourth problem used Gray-code, while the rest used standard binary code. The results for these four problems are shown in Table I. The distances between cities were generated randomly for each problem.

Table I: Traveling Salesman Problem - Average Final Tour Length

PROBLEM	MRSH1	MRSH2	MRSH2	EGA	PBIL	SGA	GA-Scale
TSP 128 (binary)	2516.3	2122.7	2144.1	1980.2	1718.2	3256.4	2275.8
TSP 200 (binary)	7815.5	7707.2	7642.3	6993.0	6097.5	12012.1	7966.8
TSP 255 (binary)	5002.5	4201.6	4381.2	4587.1	4545.4	8756.5	5598.2
TSP 255 (Gray-Code)	5054.3	3558.7	4051.0	4587.1	4484.3	8644.8	5550.5

5.2 Jobshop Scheduling Problems

Recently, genetic algorithms have been applied to the jobshop scheduling problem because it is difficult for conventional search based methods to find near-optimal solutions in a reasonable amount of time [Fang *et al*., 1993]. A good description of the jobshop problem is given by Fang:

> "In the general job shop problem, there are j jobs and m machines; each job comprises a set of tasks which must each be done on a different machine for different specified processing times, in a given job-dependent order. ... A legal schedule is a schedule of job sequences on each machine such that each job's task order is preserved, a machine is not processing two different jobs at once, and different tasks of the same job are not simultaneously being processed on different machines. The problem is to minimize the total elapsed time between the beginning of the first task and the completion of the last task (the makespan)" [Fang *et al.*, 1993].

The problem is encoded in two ways. The first encoding is derived from [Fang *et. al*, 1993]. The exact encoding can be found in [Fang *et al*., 1993] and [Baluja, 1994]. The difference between this encoding and that used by Fang is that in this study, bit strings were used to encode the integers (in standard binary encoding) in the range of 1..J. Fang used chunks which are atomic for the GA. Although the encoding used here makes the problem difficult for these optimization techniques, it is used to provide results which are comparable to other algorithms. As the makespan is to be minimized, the evaluation of the potential solution is (1.0/**makespan**). Two standard test problems are attempted, a 10 job, 10 machine problem and a 20-job, 5-machine problem. A description of the problems can be found in [Muth & Thompson, 1963]. The results are shown in Table II.

Table II: Jobshop Scheduling - Minimum Makespan - Using First Encoding.

PROBLEM	MRSH1	MRSH2	MRSH3	EGA	PBIL	SGA	GA-Scale
Jobshop 10x10	1033.0	1003.4	1003.9	984.3	979.4	1002.0	1001.2
Jobshop 20x5	1295.3	1284.6	1282.9	1204.8	1200.5	1256.3	1213.9

The second encoding is very similar to the encoding used in the Traveling Salesman Problem. The drawback of this encoding is that it uses more bits than the previous one. Nonetheless, empirically, it revealed improved results. Each job is assigned M entries of size $\log_2(J*M)$ bits. The total length of the encoding is

$J*M*\log_2(J*M)$. Therefore, each of these problems is encoded in a 700 bit solution vector. The value of each entry (of length $\log_2(J*M)$) determines the order in which the jobs are scheduled. The job which contains the smallest valued entry is scheduled first, etc. The order in which the machines are selected for each job depends upon the ordering required by the problem specification. The results are shown in Table III. With this encoding, six out of the seven algorithms perform better than, or at least as well as, the first jobshop encoding presented (the performance of MRSH-1 does not improve with this encoding).

Table III: Jobshop Scheduling - Minimum Makespan - Using Second Encoding

PROBLEM	MRSH1	MRSH2	MRSH3	EGA	PBIL	SGA	GA-Scale
Jobshop 10x10	1059.0	968.2	970.2	963.4	960.6	967.1	961.4
Jobshop 20x5	1360.6	1217.1	1215.5	1186.2	1182.0	1230.4	1190.3
Jobshop 20x5 (Randomly generated)	1164.5	1031.2	1025.5	995.0	992.1	1044.2	1005.1

5.3 Knapsack Problem

In the knapsack problem, there is a single bin of limited capacity, and **M** elements of varying sizes and values. The problem is to select the elements which will yield the greatest summed value without exceeding the capacity of the bin. The evaluation of the quality of the solution is judged in two ways: If the solution selects too many elements, such that the summed size of the elements is too large, the solution is judged by how much it exceeds the capacity of the bin - the less it exceeds the capacity, the better the solution. If the sum of the element sizes is within the capacity of the bin, the sum of the values of the selected elements is used as the evaluation. To ensure that the solutions which overfill the bin are not competitive with those which do not, their evaluations are multiplied by a small constant. This makes the invalid solutions competitive only when there are no solutions in the population which are valid. The evaluations are described below.

$$10^{-10} \times \left(\sum_{allElements} size - \sum_{selectedElements} size \right)$$

if size is greater than capacity of bin

$$\sum_{selectedElements} value$$

if size is less than or equal to capacity of bin

The weights and values for each problem were randomly generated. In the first two problems, having 512 and 2000 elements respectively, a unique element is represented by each bit. When a bit is set to 1, the corresponding element is included. In the third and fourth problems, there are 100 and 120 unique elements, respectively. However, there are 8 and 32 copies of each element; the number of elements of each type which are included in the solution is determined by interpreting a bit string, length 3 ($\log_2 8$) bits and 5 ($\log_2 32$) bits, into decimal, respectively. The results are given in Table IV. Note that the SGA algorithm was unable to find valid solutions in the second and fourth problems.

Table IV: Knapsack Problem - Average of Best Values

PROBLEM	MRSH1	MRSH2	MRSH3	EGA	PBIL	SGA	GA-Scale
Knap (512 elem., 1 cpy)	27.1	24.1	26.7	79.8	80.1	62.2	72.3
Knap (2000 elem., 1 cpy)	41.3	39.0	38.8	171.5	130.8	0.0	135.2
Knap (100 elem., 8 cpy)	148.4	136.3	101.1	394.9	403.7	49.3	414.2
Knap(120 elem., 32 cpy)	848.7	801.1	669.0	2856.4	2920.4	0.0	2559.1

5.4 Bin Packing

In the bin packing problem, there are **N** bins of varying capacities and **M** elements of varying sizes. The problem is to pack the bins with elements as tightly as possible, without exceeding the maximum capacity of any bin. In the problems attempted here, the error is measured by:

$$ERROR = \sum_{i=1}^{N} \left| CAP_i - ASSIGNED_i \right|$$

CAP_i is the capacity of bin i

$ASSIGNED_i$ is the total size of the elements in bin

The solution is encoded in a bit string of length **M** * $\log_2$**N**. Each element to be packed is assigned a sequential substring of length $\log_2$**N** whose value indicates the bin in which the element is placed. In order to minimize the **ERROR**, the evaluation of the potential solution is 1.0/**ERROR**.

Four bin packing problems of various sizes were tested: 32 bins, 128 elements; 16 bins, 128 elements; 4 bins, 256 elements; and 2 bins, 512 elements. All of the problems generated were guaranteed to have a solution with 0.0 error. The results are shown below, in Table V.

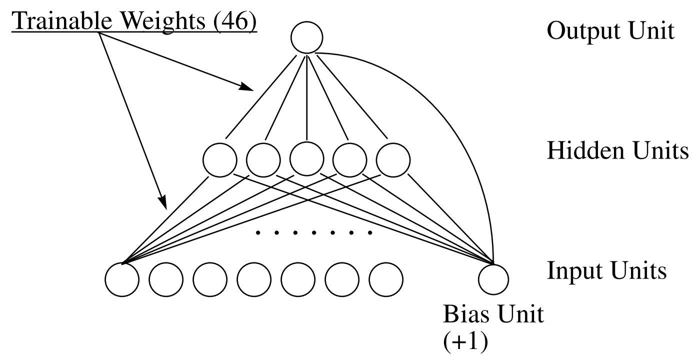
Table V: Bin Packing Problems - Minimum Error

PROBLEM	MRSH1	MRSH2	MRSH3	EGA	PBIL	SGA	GA-Scale
Bin(32 bins,128 elem.)	0.62	0.58	0.62	0.49	0.42	0.54	0.51
Bin(16 bins,128 elem.)	4.4×10^{-2}	6.8×10^{-2}	3.8×10^{-2}	3.0×10^{-2}	2.2×10^{-2}	2.8×10^{-2}	3.1×10^{-2}
Bin(4 bins, 256 elem.)	1.2×10^{-5}	1.9×10^{-5}	6.0×10^{-6}	7.0×10^{-6}	6.9×10^{-6}	7.8×10^{-6}	5.0×10^{-6}
Bin (2 bins, 512 elem.)	2.4×10^{-7}	2.4×10^{-6}	3.5×10^{-8}	4.0×10^{-8}	1.8×10^{-8}	1.2×10^{-7}	4.8×10^{-7}

5.5 Evolving Weights for an Artificial Neural Network (ANN)

Recently, evolutionary algorithms have been used to evolve the weights of artificial neural networks. In the experiments reported here, the weights of two small predefined network architectures were evolved. In the first test, the object of the neural network was to identify the parity of 7 inputs. The inputs were either 0 (represented by -0.5) or 1 (represented by 0.5). If the parity was 1, the target output is 0.5; if the parity was 0, the target output is -0.5. The evaluation was the sum of squares error on the 128 training examples. A bias input (a unit whose input is set to 1.0) was also used; this has connections to the hidden and the output units [Hertz, Krogh & Palmer, 1993]. The network architecture consisted of 8 input units (including bias), 5 hidden units, and 1 output unit. The network was fully connected between sequential layers. There were a total of 46 connections in the network, the values of the weights were restricted to the range of -10.0 to +10.0. All hidden and output units used a sigmoid activation function. Weights were represented as binary and gray code, and were assigned 8 non-overlapping bits in the solution string.

Figure 4: Network Architecture.



In the second two tests, eight real valued inputs were used. The first two inputs represent the coordinates of a point within a square with upper left corner (ULC) of (-1.0, 1.0) and lower right corner (LRC) of (1.0, -1.0). The task was to determine whether the point fell into a square region between ULC(-0.75, 0,75), and LRC (0.75,-0.75) and outside a smaller square with ULC (-0.35, 0,35), and LRC (0.35,-0.35). A diagram of this is shown in Figure 5. 5 inputs contained random noise in the region [-1:+1]. This noise was determined in the beginning of the run, and remained the same, in each training example, throughout the run. The last input was a bias unit. In total, the network had 8 inputs (including bias), 5 hidden units, and 1 output; this created 46 connections. For training, 100 uniformly distributed examples were used. The same representation and scaling of weights was used as in the previous problem. In these two problems, weights were represented as binary and Gray code, respectively. The results are shown in Table VI.

Figure 5: Training Examples for the ANN Square Problem.

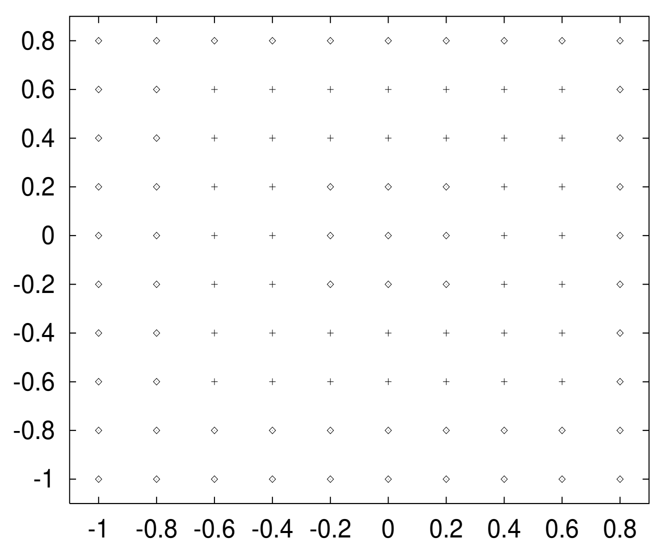


Table VI: ANN Weight Optimization - Sum of Squares Error.

PROBLEM	MRSH1	MRSH2	MRSH3	EGA	PBIL	SGA	GA-Scale
ANN PARITY 7(binary)	16.1	16.1	23.3	12.8	11.2	15.0	13.3
ANN PARITY 7(Gray)	13.7	15.1	16.4	8.3	8.2	11.6	12.3
ANN SQUARE (binary)	14.1	14.5	17.5	11.6	10.9	12.7	13.0
ANN SQUARE (Gray)	8.4	7.9	9.3	6.7	8.3	8.9	9.3

5.6 Numerical Function Optimization

In this section, the seven algorithms are compared on three numerical optimization problems. In the first and second problems, the variables in the first portions of the solution string have a large influence on the quality of the rest of the solution; small changes in their values can cause large changes in the evaluation of the solution. In the third problem, each variable can be set independently. Each variable, x_i, was represented using 9 bits, and was scaled uniformly into the range ±2.56. To avoid a division by zero error, a small constant, **C** (=0.00001), was added to the denominator of each function. Each problem was tested with the variables represented in standard binary and Gray code. Results are shown in Table VII. The maximization functions are:

$$-2.56 \le x_i < 2.56,\ i = 1 \dots 100$$

$$y_1 = x_1 \qquad y_i = x_i + y_{i-1},\ i = 2 \dots 100 \qquad f_1 = \frac{1.0}{C + \left| |y_1| + \sum_{i=2}^{100} |y_i| \right|}$$

$$y_1 = x_1 \qquad y_i = x_i + \sin\left(y_{i-1}\right),\ i = 2 \dots 100 \qquad f_2 = \frac{1.0}{C + \left| |y_1| + \sum_{i=2}^{100} |y_i| \right|}$$

$$f_3 = \frac{1.0}{C + \sum_{i=1}^{100} \left| \left((0.024 \times (i+1)) - x_i \right) \right|}$$

PROBLEM	MRSH1	MRSH2	MRSH3	EGA	PBIL	SGA	GA-Scale
F1 (Binary) (x100)	1.04	1.01	0.97	1.93	2.12	1.96	1.72
F1 (Gray Code)(x100)	1.21	1.18	1.17	2.06	2.62	1.92	1.78
F2 (Binary) (x100)	3.08	3.06	2.91	4.00	4.40	3.58	3.68
F2 (Gray Code) (x100)	4.34	4.38	4.28	4.67	5.61	3.64	4.63
F3 (Binary) (x 100)	8.07	8.10	7.56	14.57	16.43	9.171	12.30
F3 (Gray Code) (x100)	416.64	416.64	416.64	331.69	366.77	28.35	210.37

6. SUMMARY OF EMPIRICAL RESULTS

Many results have been presented in the previous section. This paper has concentrated on breadth; a large number of problems were attempted with seven optimization heuristics. It should be noted that because all of the algorithms have tunable parameters, it is possible that different settings may yield different results. Additional mechanisms, which take advantage of problem specific information, may also improve the performance of each of these methods. Nonetheless, by selecting a variety of problems and problem sizes to compare, all of the algorithms should show their strengths and weaknesses in some portion of the test set.

The relative ranks of the algorithms on all of the problems are shown in Table VII; this table ranks the algorithms with respect to the average best results produced over all runs. It should be noted that with only 20 runs per algorithm, not all of the differences are statistically significant. More details on the differentials between each algorithm's performance were presented in Section 5. In terms of the final solutions found, the PBIL algorithm worked the best overall, followed by the EGA algorithm. In the majority of problems attempted here (25 out of 27), learning from negative examples improved the quality of the final solutions found (PBIL performed better than EGA). Only in two of the problems did the negative learning hurt the performance of the PBIL algorithm (EGA performed better than PBIL).

In terms of clock speed, the MRSH algorithms worked the fastest. However, if the time for each chromosome/solution string's evaluation is much larger than the time for the algorithm's procedures, this benefit diminishes. Moves to equal regions (rather than only strictly better regions) had mixed results overall. Nonetheless, in several problems, such as the jobshop (both encodings) and TSP problems, the moves to equal regions improved performance. In other problems sets, not explored here, it was also found that moves to regions of equal evaluations were important for good performance [Juels & Wattenberg, 1994]. MRSH did well on the largest TSP examined; it was able to find a shorter tour than the other algorithms. (when encoded in binary and Gray Code). Similarly, in F3, MRSH was able to take the largest advantage of the gray code.

Although the standard genetic algorithm (SGA) performed only as well as the MRSH algorithms, the GA-scale algorithm performed slightly better. A summary of the results can be found in Table VIII. This table has the following columns:

1. In the cases in which PBIL did better than GA-Scale, this column gives the generation in which PBIL was able, on average, to surpass the highest evaluation GA-Scale found, on average, in its 20 runs. For example, in the first problem: TSP-128 (binary), the highest evaluation of the GA was 2275.8 (Table I), by generation 210, PBIL was able to surpass this evaluation.

2. The same numbers are given for GA-Scale. For example, on the knapsack01 (2000 elem, 1 copy) problem, the highest evaluation PBIL was able to obtain was 403.7, in generation 1505 GA-Scale was, on average, able to surpass it.

Columns (3),(4) and (5) of Table VIII compare the three different types of algorithms:

3. Marks the problems on which any form of MRSH was able to do better than GA-Scale.

4. Marks when any form of MRSH did better than PBIL.

5. Marks the problems in which GA-Scale outperformed PBIL.

Columns 6,7 and 8 of Table VIII compare different variations of each algorithm.

6. Indicates the problems on which moves to equal regions helped the performance of MRSH. These are the problems in which either MRSH-2 or MRSH-3 did better than MRSH-1.

7. Marks the problems in which learning from negative examples helped the PBIL/EGA algorithm. These are the problems in which PBIL performed better than EGA.

8. Indicates the problems in which GA-Scale did better than SGA. The improvement may be due to the scaling of fitness values and/or the different crossover operators (GA-Scale: Uniform, SGA: Two Point).

For the problems which were attempted with gray and binary code, Table IX provides a list of which algorithms benefited from using gray code.

7. CONCLUSIONS

This paper has presented results on many problems. From the results reported in this paper, it is evident that algorithms which are simpler than standard GAs can perform comparably to GAs, on both small and large problems. Other studies have also shown this for various sets of problems [Juels & Wattenberg, 1994][Mitchell & Forrest, 1992], etc. In studies analyzing the performance of GAs on particular problems, these results suggest that analyses should include comparisons not only to other GAs, but also to other simpler methods of optimization before a benefit is claimed in favor of GAs. This study did not include techniques such as Simulated Annealing or Tabu Search, which should be included in the future.

It is interesting to note that the PBIL algorithm, which does not use the crossover operator, and redefines the role of the population to one which is very different than that of a GA, performs either better than or comparably to a GA on the majority of the problems. PBIL and GAs both generate new trials based on statistics from a population of prior trials. The PBIL algorithm explicitly maintains these statistics, while the GA implicitly maintains them in its population. The GA extracts the statistics by the selection and crossover operators. More detailed comparisons between these two algorithms can be found in [Baluja & Caruana, 1995].

It should be noted that the relative performance of GAs in comparison to PBIL will improve as the population size of the GA increases [Baluja & Caruana, 1995]. As the population size of the GA increases, the GA will be able to maintain more dissimilar points in its population, and will therefore be able to use crossover more effectively. On the other hand, in its current implementation, the PBIL algorithm only use a few solution vectors for updating the probability vector regardless of the population size. Nonetheless, the large population size needed by a GA is not always feasible because of the need to balance the number of generations required and the total number of evaluations possible. However, even when the resources to use large populations are available, a large amount of empirical work has shown that using an “parallel island-model” GA may be more effective than a single panmictic population. The island-model evolves multiple small population in parallel, with only a small amount of interaction between subpopulations. The parallel subpopulation model, in many cases, has outperformed the use of a single large population; see for example: [Davidor *et. al*, 1993][Gordon & Whitley, 1993][Baluja, 1993][Whitley & Starkweather, 1990]. If these parallel subpopulations are used instead of a single large population, each subpopulation can be modeled with individual probability vectors, as in the PBIL algorithm.

A GA with different mechanisms, such as non-stationary mutation rates, local optimization heuristics, parallel subpopulations, specialized crossover, or larger operating alphabets, may perform better than the GAs explored here. It should be noted, however, that all of these mechanisms, with the exception of specialized crossover operators, can be used with PBIL with few, if any, modifications.

Another direction to explore in the future is how these algorithm perform with alternate solution encodings; in this study only binary encodings were used. Although work has already been conducted in this area with GAs (a good introduction to this can be found in [Eshelman & Schaffer, 1992]), how well will PBIL or MRSH perform with these alternate encodings? Finally, in this study, optimization was only explored in static environments. Future research should also include search and optimization in dynamic environments, or environments which require maximization of cumulative payoff. The adaptive nature of GAs may reveal a pronounced benefit in these more complex domains.

Table VII: Summary of Ranks

	Encoding Length (bits)	MRSH1	MRSH2	MRSH3	EGA	PBIL	SGA	GA-Scale	MRSH(1,2,3) BEST	PBIL/EGA BEST	SGA/GA-SCALE BEST
TSP 128 (binary)	896	6	3	4	2	1	7	5		●	
TSP 200 (binary)	1600	5	4	3	2	1	7	6		●	
TSP 255 (binary)	2040	5	1	2	4	3	7	6	●		
TSP 255 (Gray-Code)	2040	5	1	2	4	3	7	6	●		
Jobshop 10x10 (Encoding 1)	500	7	5	6	2	1	4	3		●	
Jobshop 20x5 (Encoding 1)	500	7	6	5	2	1	4	3		●	
Jobshop 10x10 (Encoding 2)	700	7	5	6	3	1	4	2		●	
Jobshop 20x5 (Encoding 2)	700	7	5	4	2	1	6	3		●	
Jobshop 20x5 (Encoding 2)	700	7	5	4	2	1	6	3		●	
Knap (512 elem., 1 copy)	512	5	7	6	2	1	4	3		●	
Knap (2000 elem., 1 copy)	2000	4	5	6	1	3	7	2		●	
Knap (100 elem., 8 copies)	300	4	5	6	3	2	7	1			●
Knap (120 elem., 32 copies)	600	4	5	6	2	1	7	3		●	
Bin (32 bins, 128 elem.)	640	6	5	6	2	1	4	3		●	
Bin (16 bins, 128 elem.)	512	6	7	5	3	1	2	4		●	
Bin (4 bins, 256 elem.)	512	6	7	2	4	3	5	1			●
Bin (2 bins, 512 elem.)	512	5	7	2	3	1	4	6		●	
ANN PARITY 7 (binary)	368	5	5	7	2	1	4	3		●	
ANN PARITY 7 (gray)	368	5	6	7	2	1	3	4		●	
ANN SQUARE (binary)	368	5	6	7	2	1	3	4		●	
ANN SQUARE (gray)	368	4	2	7	1	3	5	6		●	
F1 (Encoded in Binary)	900	5	6	7	3	1	2	4		●	
F1 (Encoded in Gray Code)	900	5	6	7	2	1	3	4		●	
F2 (Encoded in Binary)	900	5	6	7	2	1	4	3		●	
F2 (Encoded in Gray Code)	900	5	4	6	2	1	7	3		●	
F3 (Encoded in Binary)	900	6	5	7	2	1	4	3		●	
F3 (Encoded in Gray Code)	900	1	1	1	5	4	7	6	●		
TOTAL (27 Problems)									3	22	2

Table VIII: Comparison of Methods

	GENERATION IN WHICH PBIL EXCEEDED BEST GA-SCALE	GENERATION IN WHICH GA-SCALE EXCEEDED BEST PBIL	MRSH{1,2,3} BETTER THAN GA-SCALE	MRSH{1,2,3} BETTER THAN PBIL	GA-SCALE BETTER THAN PBIL	MOVES TO EQUAL VALUE REGIONS BENEFICIAL (MRSH)	IMPROVED WITH "NEGATIVE" LEARNNIG RATE (EGA-PBIL)	GA-SCALE BETTER THAN SGA
TSP 128 (binary)	210	-	●			●	●	●
TSP 200 (binary)	496	-	●			●	●	●
TSP 255 (binary)	772	-	●	●		●	●	●
TSP 255 (Gray-Code)	741	-	●	●		●	●	●
Jobshop 10x10 (Encoding 1)	215	-				●	●	●
Jobshop 20x5 (Encoding 1)	191	-				●	●	●
Jobshop 10x10 (Encoding 2)	1690	-				●	●	●
Jobshop 20x5 (Encoding 2)	418	-				●	●	●
Jobshop 20x5 (Encoding 2)	366	-				●	●	●
Knap (512 elem., 1 copy)	277	-					●	●
Knap (2000 elem., 1 copy)	-	1505			●			●
Knap (100 elem., 8 copies)	-	633			●		●	●
Knap (120 elem., 32 copies)	700	-					●	●
Bin (32 bins, 128 elem.)	1039	-				●	●	●
Bin (16 bins, 128 elem.)	235	-				●	●	
Bin (4 bins, 256 elem.)	-	1491		●	●	●	●	●
Bin 2 bins, 512 elem.)	968	-	●			●	●	
ANN PARITY 7 (binary)	500	-					●	●
ANN PARITY 7 (gray)	650	-					●	
ANN SQUARE (binary)	250	-					●	
ANN SQUARE (gray)	1000	-	●	●		●		
F1 (Encoded in Binary)	630	-					●	
F1 (Encoded in Gray Code)	512	-					●	
F2 (Encoded in Binary)	492	-					●	●
F2 (Encoded in Gray Code)	1272	-				●	●	●
F3 (Encoded in Binary)	429	-				●	●	●
F3 (Encoded in Gray Code)	947	-	●	●			●	●
TOTAL (27 Problems)	-	-	7	5	3	16	25	20

Table IX: Algorithms which benefited by using Gray Code over standard binary code.

	MRSH1	MRSH2	MRSH3	EGA	PBIL	SGA	GA-Scale
TSP 255		●	●		●	●	●
ANN PARITY 7	●	●	●	●	●	●	●
ANN SQUARE	●	●	●	●	●	●	●
F1	●	●	●	●	●		●
F2	●	●	●	●	●	●	●
F3	●	●	●	●	●	●	●

ACKNOWLEDGEMENTS

Thanks are extended to Henry Rowley, Conrad Poelman, Rich Caruana, and Kaari Flagstad who all help proof-read different versions of this paper and provided useful suggestions and discussions. The impetus and foundations for this work came from the collaboration with Ari Juels, at the University of California - Berkeley.

REFERENCES

Ackley, D.H. (1987) "An Empirical Study of Bit Vector Function Optimization" in Davis, L. (ed) *Genetic Algorithms and Simulated Annealing*, 1987. Morgan Kaufmann Publishers, Los Altos, CA.

Baluja, S. (1993) "The Evolution of Genetic Algorithms: Towards Massive Parallelism". In Utgoff, P. (ed) *Proceedings of the Tenth International Conference on Machine Learning*, pp. 1-8. Morgan Kaufman Publishers.

Baluja, S. (1994) "Population-Based Incremental Learning: A Method for Integrating Genetic Search Based Function Optimization and Competitive Learning". Carnegie Mellon University. Technical Report. CMU-CS-94-163. Available via anonymous FTP at reports.adm.cs.cmu.edu.

Baluja, S. & Caruana, R. (1995) "Removing the Genetics from the Standard Genetic Algorithm", in A. Prieditis & S. Russel (ed.) *Machine Learning: Proceedings of the Twelfth International Conference.* Morgan Kaufmann Publishers, San Francisco, CA. Report also available via anonymous FTP at reports.adm.cs.cmu.edu (CMU-CS-95-141).

Cobb, H. (1992) "Genetic Algorithms for Tracking Changing Environments", in Forrest (ed). *(ICGA-5) Proceedings of the Fifth International Conference on Genetic Algorithms*. 523-530. Morgan Kaufmann Publishers. San Mateo, CA.

Cohn, D.A. (1994) "Neural Network Exploration Using Optimal Experiment Design" in Cowan, Tesauro, Alspector (eds.) (NIPS-6). *Advances in Neural Information Processing Systems.* 679-686. Morgan Kaufmann Publishers. San Mateo, CA.

De Jong, K. (1975) *An Analysis of the Behavior of a Class of Genetic Adaptive Systems*. Ph.D. Dissertation.

De Jong, K. (1992) "Genetic Algorithms are NOT Function Optimizers". In Whitley (ed.) *FOGA-2 Foundations of Genetic Algorithms-2*. 5-17. Morgan Kaufmann Publishers. San Mateo, CA.

Davis, L. (1991) "Bit-Climbing, Representational Bias, and Test Suite Design", *Proceedings of the Fourth International Conference on Genetic Algorithms*. (18-23). Morgan Kaufmann Publishers. San Mateo, CA

Davidor, Y., Yamada, T., Nakano, R. (1993) "The ECOlogical Framework II: Improving GA Performance at Virtually Zero Cost". In Forrest, S. (ed) *ICGA-5.Proceedings of the Fifth International Conference on Genetic Algorithms*. 171-176.

Eshelman, L.J. & Schaffer, D. (1992) "Real-Coded Genetic Algorithms and Interval Schemata" in *FOGA-2 Foundations of Genetic Algorithms-2*. Morgan Kaufmann Publishers. San Mateo, CA187-202.

Eshelman, L.J. & Schaffer, D. (1993) "Crossover's Niche". In Forrest (ed). *(ICGA-5) Proceedings of the Fifth International Conference on Genetic Algorithms*. 9-14. Morgan Kaufmann Publishers. San Mateo, CA.

Fang, H.L, Ross, P., Corne, D. (1993) "A Promising Genetic Algorithm Approach to Job-Shop Scheduling, Rescheduling, and Open- Shop Scheduling Problems". In Forrest, S. (ed) *ICGA-5.Proceedings of the Fifth International Conference on Genetic Algorithms*.

Forrest, S. and Mitchell, M (1993) "What Makes a Problem Hard for a Genetic Algorithm" *Machine Learn-*

ing 13. 2 & 3. Kluwer Academic Publishers, Boston MA.

Goldberg, D.E. (1989) *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley.

Gordon, V.S. & Whitley, D. (1993) "Serial and Parallel Genetic Algorithms as Function Optimizers," In Forrest, S. (ed) *ICGA-5.Proceedings of the Fifth International Conference on Genetic Algorithms*. 177-184.

Grefenstette, J.J. & Fitzpatrick, J.M. (1988) "Genetic Algorithms in Noisy Environments" Machine Learning, Vol. 3 No. 2/3, Kluwer Academic Publishers, pp. 101-120.

Hertz, J., Krogh, A., & Palmer, G. (1993). *Introduction to the Theory of Neural Computation.* Addison-Wesley.

Holland, J. H. Adaptation in Natural and Artificial Systems. Ann Arbor: The University of Michigan Press.

Juels, Ari. (1993, 1994) Personal Communication. University of California - Berkeley.

Koza, J.R. (1992) Genetic Programming: On the Programming of Computers by Means of Natural Selection. Cambridge, MA: The MIT Press.

Langton, C. (1994) Artificial Life Journal. MIT Press.

Mitchell, M. and Holland, J. (1994) "When will a Genetic Algorithm Outperform Hill Climbing" *Advances in Neural Information Processing Systems 6*, 1994. Cowan, Tesauro, Alspector (eds). Morgan Kaufmann Publishers. San Francisco, CA.

Muth & Thompson (1963) *Industrial Scheduling* Prentice Hall International. Englewood Cliffs, NJ.

Syswerda, G. (1989) "Uniform Crossover in Genetic Algorithms". In *Proceedings of the Third International Conference on Genetic Algorithms and their Applications*, 2-9. J.D. Schaeffer, ed. Morgan Kaufmann.

Syswerda, G. (1992) "Simulated Crossover in Genetic Algorithms". *FOGA-2*. 239-255. *FOGA-2 Foundations of Genetic Algorithms-2*. Morgan Kaufmann Publishers. San Mateo, CA.

Wattenberg, M. & Juels, A. (1994) "Stochastic Hillclimbing as a Baseline Method for Evaluating Genetic Algorithms," University of California - Berkeley, Technical Report. CSD-94-834.

Whitley, D. & Starkweather, T. "GENITOR II: a distributed genetic algorithm". Journal of Experimental and Theoretical Artificial Intelligence 2. 189-214.