Population-Based Incremental Learning:

A Method for Integrating Genetic Search Based Function Optimization and Competitive Learning

Shumeet Baluja

June 2, 1994
CMU-CS-94-163

School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

Abstract

Genetic algorithms (GAs) are biologically motivated adaptive systems which have been used, with varying degrees of success, for function optimization. In this study, an abstraction of the basic genetic algorithm, the Equilibrium Genetic Algorithm (EGA), and the GA in turn, are reconsidered within the framework of competitive learning. This new perspective reveals a number of different possibilities for performance improvements. This paper explores population-based incremental learning (PBIL), a method of combining the mechanisms of a generational genetic algorithm with simple competitive learning. The combination of these two methods reveals a tool which is far simpler than a GA, and which out-performs a GA on large set of optimization problems in terms of both speed and accuracy. This paper presents an empirical analysis of where the proposed technique will outperform genetic algorithms, and describes a class of problems in which a genetic algorithm may be able to perform better. Extensions to this algorithm are discussed and analyzed. PBIL and extensions are compared with a standard GA on twelve problems, including standard numerical optimization functions, traditional GA test suite problems, and NP-Complete problems.

Shumeet Baluja is supported by a National Science Foundation Graduate Fellowship. 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 or the U.S. government.

KEYWORDS:

Genetic Algorithms, Supervised Competitive Learning, Equilibrium Genetic Algorithm, Learning Vector Quantization.

Population-Based Incremental Learning:

A Method for Integrating Genetic Search Based Function Optimization and Competitive Learning

Shumeet Baluja
baluja@cs.cmu.edu
School of Computer Science
Carnegie Mellon University

Abstract:

Genetic algorithms (GAs) are biologically motivated adaptive systems which have been used, with varying degrees of success, for function optimization. In this study, an abstraction of the basic genetic algorithm, the Equilibrium Genetic Algorithm (EGA), and the GA in turn, are reconsidered within the framework of competitive learning. This new perspective reveals a number of different possibilities for performance improvements. This paper explores population-based incremental learning (PBIL), a method of combining the mechanisms of a generational genetic algorithm with simple competitive learning. The combination of these two methods reveals a tool which is far simpler than a GA, and which out-performs a GA on large set of optimization problems in terms of both speed and accuracy. This paper presents an empirical analysis of where the proposed technique will outperform genetic algorithms, and describes a class of problems in which a genetic algorithm may be able to perform better. Extensions to this algorithm are discussed and analyzed. PBIL and extensions are compared with a standard GA on twelve problems, including standard numerical optimization functions, traditional GA test suite problems, and NP-Complete problems.

Keywords:

Genetic Algorithms, Competitive Learning, Equilibrium Genetic Algorithm, Learning Vector Quantization.

1. Introduction

Genetic algorithms and artificial neural networks are two recently developed techniques that have received a large amount of attention in the artificial intelligence research communities. Many of the current efforts in genetic algorithm research have stressed their role as general purpose function optimizers, which have been applied to functions in the fields of biological modelling, NP-complete problem approximation heuristics and standard numerical optimization. A large part of the research effort in the field of neural networks and competitive learning has been devoted to clustering and classification tasks, with a wide variety of applications, such as feature mapping, speech recognition [Waibel, 1989], scene analysis and robot control [Pomerleau, 1989] and data compression [Hertz, Krogh, Palmer, 1993]. There are also many continuing efforts to perform classification and concept learning tasks using genetic algorithms [De Jong et al., 1993][Janikow, 1993][Booker et al., 1990][Goldberg,1989]. Similarly, optimization using artificial neural networks is a widely researched area. Perhaps the most widely known attempt is that of the Hopfield and Tank Traveling Salesman model [Hopfield & Tank, 1985][Hertz,Krogh, Palmer, 1993].

An abstraction of the basic GA, termed the "Equilibrium Genetic Algorithm" (EGA), has been developed in conjunction with [Juels, 1993,1994]. The EGA attempts to describe the limit population of a genetic algorithm by an equilibrium point, under the assumption that the population is always "mated to convergence." This process can be viewed as a form of eliminating the explicit crossover step in standard genetic algorithm search. In the work with Juels, the representation of the population by using an equilibrium point was explored. The work presented in this paper concentrates on the operators needed to make this new representation effective for search; the operators are presented in the context of supervised competitive learning. Issues regarding the use of the resulting algorithm, such as the settings of the parameters, the contribution of a mutation operator, as well as the algorithm's relation to more traditional genetic search, are examined. It is determined that the framework of competitive learning provides a basis from which to explain the performance of the EGA. The new framework also provides the mechanisms for many extensions to the basic algorithm which improve the performance of the EGA. [1] Several of the extensions are explored in this paper, and many more are presented as directions for future research.

This paper presents population-based incremental learning (PBIL), a method of combining genetic algorithms and competitive learning for function optimization. PBIL is an extension to the EGA algorithm achieved through the re-examination of the performance of the EGA in terms of competitive learning. This paper also presents a detailed empirical examination of the EGA and PBIL algorithms on a suite of problems designed to test their abilities in a wide variety of function optimization domains. Although this work has ties with other areas of research involving genetic algorithms, to limit the scope of this paper, PBIL is explored exclusively in the context of function optimization. Other recent work in combining GAs with techniques related to artificial neural networks has been largely directed in alternative directions, such as using GAs for optimizing neural network structure and performance [Whitley & Schaffer, 1992][Gruau, 1993][Polani & Uthamann, 1993][Whitley & Hanson, 1989]. One of the closest related works, which also employed an underlying artificial neural network structure to augment genetic search, can be found in Ackley's *A Connectionist Machine for Genetic Hillclimbing*, [Ackley, 1987][Ackley, 1985].

The remainder of this section describes the principles of genetic based optimization and competitive learning. Section 2 describes the combination of both of the techniques by examining the role of the population in a genetic algorithm. Section 3 examines the effects of changing some of the parameters in the PBIL algorithm. Section 4 presents extensions of the algorithm, based on Kohonen's learning vector quantization algorithm [Kohonen, 1988,1989]. In section 5, the basic PBIL and its extensions are empirically examined on a suite of twelve test problems. Section 6 discusses the applicability of the results reported in section 5 and concludes with a discussion of the benefits of using PBIL in comparison to standard genetic algorithms for function optimization. Finally, section 7 closes this report with suggestions for future research.

1.1. Introduction to Genetic Algorithms

Genetic algorithms (GAs) are biologically motivated adaptive systems which are based upon the principles of natural selection and genetic recombination. Although GAs have often been used for function optimization, some of the pioneering work [Holland, 1975] suggests an alternative view of GA behavior. De Jong summarizes this view in the following paragraph:

> "Holland [Holland, 1975] has suggested that a better way to view GAs is in terms of optimizing a sequential decision process involving uncertainty in the form of lack of a priori knowledge, noisy feedback and a time-varying payoff function. More specifically, given a limited number of trials, how should one allocate them so as to maximize the cumulative payoff in the face of all this uncertainty?" [De Jong, 92].

This has important implications to the use of genetic algorithms as function optimizers. Although genetic algorithms may have the ability to quickly find regions of high performance in the face of noise and time-

1. To avoid confusion, the term "EGA" will be used only when referencing [Juels, 1993] explanation of the algorithm.

varying payoff functions, they may be unable to find the absolute optimal solution, in time-varying or stationary environments.

The GAs described in the current literature are highly modified to reconcile the differences between function optimization, in which the effectiveness is measured by the best solution found, and maximizing cumulative returns, in which finding the absolute best solution may not be the only critical issue. The basic genetic algorithm and some of the common modifications which have been made for effective function optimization are described below.

A GA combines the principles of survival of the fittest with a randomized information exchange. Although the information exchange is randomized, the GA is far different than a simple random walk. A GA has the ability to recognize trends toward optimal solutions, and exploit such information by guiding the search toward them.

A genetic algorithm maintains a population of potential solutions to the objective function being optimized. The initial group of potential solutions is determined randomly. These potential solutions, called "chromosomes," are allowed to evolve over a number of generations. At every generation, the fitness of each chromosome is calculated. The fitness is a measure of how well the potential solution optimizes the objective function. The subsequent generation is created through a process of selection and recombination. The chromosomes are probabilistically chosen for recombination based upon their fitness; this is a measure of how well the chromosomes achieve the desired goal (e.g. find the minimum in a specified function, etc.). The recombination operator combines the information contained within pairs of selected "parents", and places a mixture of the information from both parents into a member of the subsequent generation. Selection and recombination are the mechanisms through which the population "evolves." Although the chromosomes with high fitness values will have a higher probability of being selected for recombination than those which do not, they are not guaranteed to appear in the next generation. The "children" chromosomes produced by the genetic recombination are not necessarily better than their "parent" chromosomes. Nevertheless, because of the selective pressure applied through a number of generations, the overall trend is towards better chromosomes.

In order to perform extensive search, genetic diversity must be maintained. When diversity is lost, it is possible for the GA to settle into a sub-optimal state. There are two fundamental mechanisms which the basic GA uses to maintain diversity. The first, mentioned above, is a probabilistic scheme for selecting which chromosomes to recombine. This ensures that information other than that represented in the best chromosomes appears in the subsequent generation. Recombining only good chromosomes will very quickly converge the population without extensive exploration, thereby increasing the possibility of finding only a local optimum. The second mechanism is mutation; mutations are used to help preserve diversity and to escape from local optima. Mutations introduce random changes into the population.

The genetic algorithm is typically allowed to continue for a fixed number of generations. At the conclusion of the specified number of generations, the best chromosome in the final population, or the best chromosome ever found, is returned. Unlike the majority of other search heuristics, genetic algorithms do not work from a single point in the function space. GAs continually maintain a population of points from which the function space is explored.

In using GAs for function optimization, many issues, such as proper scaling of functions, ensuring that good information is not lost due to random chance, and efficient problem representation, need to be resolved. Although GAs can often find regions of high performance, it is much harder for the GA to move to the optimal solution. One potential reason for this inability is that the differential between good and optimal solutions may be very small in comparison with the differential between good and bad solutions. In designing an effective genetic based function optimizer, it is necessary to be able to provide a large enough "incentive" for the GA to make progress given only small differentials. One method of maintaining a large differential between potential solutions is to employ a method of dynamic scaling, so that the fitness of each solution is measured relative to the fitness of the other solutions in the current population.

As a genetic algorithm is randomized, it is possible to lose the best solution due to random chance. There is no guarantee that the best solution in the current population will be selected for recombination, or that if it is selected, that the mutation and crossover operators will not destroy some of its information as it is passed to its successors. If the best solution is lost from the population, there is no guarantee that the solution will be found again. Methods such as elitist strategies have been proposed to address this problem. Elitist strategies ensure that the best solution in the previous population is transferred to the current generation unaltered. A common implementation of this replaces the worst chromosome in the current generation with the best from the previous generation.

Beyond the mechanisms inherent to the GA which influence its ability to optimize functions, there is also the issue of problem representation. Although the majority of GA research has been conducted using a binary solution representation, this is not the only method of encoding problem solutions. Different methods which alter the cardinality of the alphabet used for encoding, or the interpretation of the encoding, can have an enormous impact on the performance of the GA [Liepins & Vose, 1990]. A good overview of the issues and difficulties involved in using GAs for function optimization can be found in [De Jong, 1992].

1.2. Introduction to Competitive Learning

Competitive learning (CL) is often used to cluster a number of unlabeled points into distinct groups. Membership into each group is based upon the similarity of points with respect to the characteristics in study. The procedure is unsupervised, as there is no *a priori* knowledge of how many groups exist, or what each group's distinguishing features may be. The hope is that the CL procedure will be able to determine the most relevant features for class formation and then be able to cluster points into distinct groups based on these features.

Competitive learning is often studied in the context of artificial neural networks as it is easily modeled in this form. A typical competitive learning network is shown in Figure 1. The inputs correspond to the feature vector for each point. The outputs correspond to the class in which the network has placed the point. In this network, there are two types of connections, excitatory and inhibitory. The inhibitory connections, between output units, ensure that only one output is turned on at a time. The output unit that is turned on is the one which has the largest net input. The excitatory connections contribute to the net input of the outputs. The algorithm used to train the network is described below.

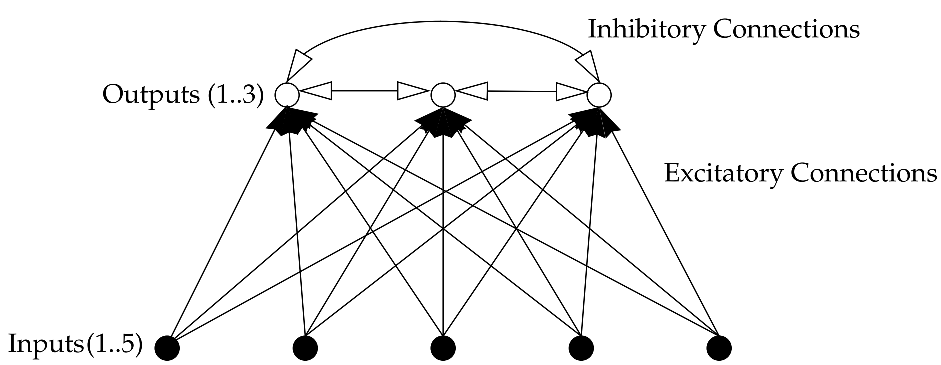


Figure 1: A competitive learning network. The hollow arrows represent inhibitory connections, the filled arrows represent excitatory connections.

The initial weights of the network are chosen randomly, and are subject to normalization constraints, which are detailed in [Hertz, Krogh & Palmer, 1993]. The activation of the output units is calculated by the following formula (in which w is the weight of the connection between i and j):

$$output_i = \sum_j w_{ij} \times input_j$$

In a competitive learning network, only the output unit with the largest activation is allowed to fire for each point presented. The winning output unit corresponds to the classification of the input point. During training, the weights of the winning output unit are moved closer to the presented point by adjusting the weights according to the following rule (LR is the learning rate parameter):

$$\Delta w_{ij} = LR \times (input_j - w_{ij})$$

The process of training the CL network involves repeatedly presenting each point to the network until the network has stabilized. Although, in general, the network cannot be guaranteed to stabilize if weight updates are made after each point is presented to the network, other heuristics may be used to determine when to stop training. One possible method of ensuring stability is to reduce the learning rate gradually to 0, as the total number of pattern presentation increases [Hertz, Krogh, Palmer, 1993].

After the network training is complete, the weight vectors for each of the output units can be considered prototype vectors for one of the discovered classes. The attributes with the large weights are the defining characteristics of the class represented by the output. It is the notion of creating a prototype vector which will be central to the discussions of PBIL. The unsupervised nature of the algorithm will not be maintained, as the members of the class of interest will be easily determinable; this will be returned to in much greater detail in the next section. Although the method of training described in this section has been unsupervised, supervised competitive learning has been explored in artificial neural network literature, and is central to the discussion in Section 2 [Kohonen et. al, 1988] [Kohonen, 1989]. The examination of PBIL through the perspective of supervised competitive learning yields insights which can lead to a much more efficient algorithm than the base-line PBIL described in the next section. Improved versions of PBIL are described in section 4, and are empirically examined in section 5.

2. Examining the Genetic Algorithm: The Role of a Population

One of the fundamental attributes of the genetic algorithm is its ability to search the function space from multiple points in parallel. In this context, parallelism does not refer to the ability to parallelize the implementation of genetic algorithms; rather, it refers to the ability to represent a very large number of potential solutions in the population of a single generation. This section describes the implicit parallelism of GAs, and explicit methods of ensuring parallelism. Because of the failure of the implicit parallelism to exist in the latter parts of genetic search in the simple genetic algorithm, the utility of maintaining multiple points, through the use of a population, decreases. The limited effectiveness of the population in the latter portions of search allows it to be modeled by a probability vector, specifying the probability of each position containing a particular value. This concept is central to the PBIL algorithm, and is the focus of this section.

2.1. Implicit and Explicit Parallelism in Genetic Search

In every generation during the run of a GA, a population of potential solutions exists. The search of the function space progresses from these points, and the schemata represented in these points, in parallel. This is in contrast to other search techniques, such as simulated annealing [Kirkpatrick et al., 1983], or hill-climbing, in which a single point in the function space is used as the basis for search. The ability to search multiple schemata in each solution vector has been termed implicit parallelism. However, useful parallelism, at the level of the population, is not easily maintained. It is possible for the population to converge to very similar solution vectors. Once the population has converged, the ability for crossover operators to aid in exploring new portions of the function space is greatly hindered. For a review of typical crossover operators see Appendix A. Premature convergence of a population can occur when the population becomes too homogenous. As the GA allocates an exponentially increasing number of trials to better solutions [Goldberg, 1989], the entire population may come to be dominated by very similar solution vectors when several consecutive generations do not develop novel high evaluation solution vectors.

In a traditional GA, the problem of premature convergence and the trap of local minima has been partially addressed by the mutation operator. The mutation operator inserts random changes into the population, without regard to whether the effects are beneficial or detrimental. However, other mechanisms, which help to maintain the parallelism explicitly, have been proposed to address the problem of diversity loss and maintaining parallelism in search [Eschelman, 1991] [Goldberg, 1989] [De Jong, 1975] [Goldberg, 1987]. To demonstrate the importance of maintaining parallelism in genetic search, the technique presented here to implement explicit parallelism is subpopulation evolution.

One method of implementing explicit parallelism is through models of genetic algorithms often referred to as "island models" or "coarse/fine grain parallel GAs" etc. The underlying premise of these models is that although genetic search often loses the parallelism inherent in a single large population structure, it is possible to maintain parallelism through the use of multiple subpopulations. In this model, the single large population of the traditional genetic algorithm is divided into many smaller subpopulation. Each subpopulation evolves its chromosomes primarily independently of the other subpopulations. Interaction, in the form of chromosome swapping between subpopulations, is restricted, and is based upon temporal and spatial considerations [Whitley, 1991] [Whitley, 1992] [Baluja, 1992] [Muhlenbien, 1989] [Schleuter, 1990].

The motivation of explicit parallelism is largely based upon the tenets of the theory of punctuated equilibria. The theory of punctuated equilibria, and its relation to GAs is described in [Cohoon et al., 1988]:

> Punctuated Equilibria is based upon two principles: allopatric speciation and stasis. Allopatric speciation involves the rapid evolution of new species after a small set of members of species, peripheral isolates, becomes segregated into a new environment. Stasis, or stability, of a species, is simply the notion of lack of change. It implies that after equilibria is reached in an environment, there is very little drift away from the genetic composition of species. ... Punctuated Equilibria stresses that a powerful method for generating new species is to thrust an old species into a new environment, where change is beneficial and rewarded. For this reason we should expect a genetic algorithm approach based upon punctuated equilibria to perform better than the typical single environment scheme [Cohoon et al, 1988].

An example which clearly displays the benefits of using a parallel GA in place of a traditional GA is shown below. The inability of a traditional genetic algorithm to maintain simultaneously multiple equally good points in the function space prevents the GA from finding the global optima. As the parallel genetic algorithm (pGA) model is able to maintain multiple good points, the pGA is able to solve the problems more consistently. This problem is attempted with a single population GA with 100 members, and an "island" model GA with 5 populations, each consisting of 20 members. The problem and results are shown below, see Figure 2. This test problem was created jointly with Juels [Juels, 1993].

$$f(\vec{x}) = MAX(o(\vec{x}), z(\vec{x})) + REWARD$$

$$REWARD = \begin{cases} (o(\vec{x}) \geq T) \wedge (z(\vec{x}) \geq T) \Rightarrow LENGTH \\ 0 \end{cases}$$

LENGTH is the number of bits in the solution string
o(x) is the number of contiguous ones starting at the first position
z(x) is the number of contiguous zeros ending at the last position
T is the threshold, less than LENGTH/2
MAX returns the larger of its two arguments.

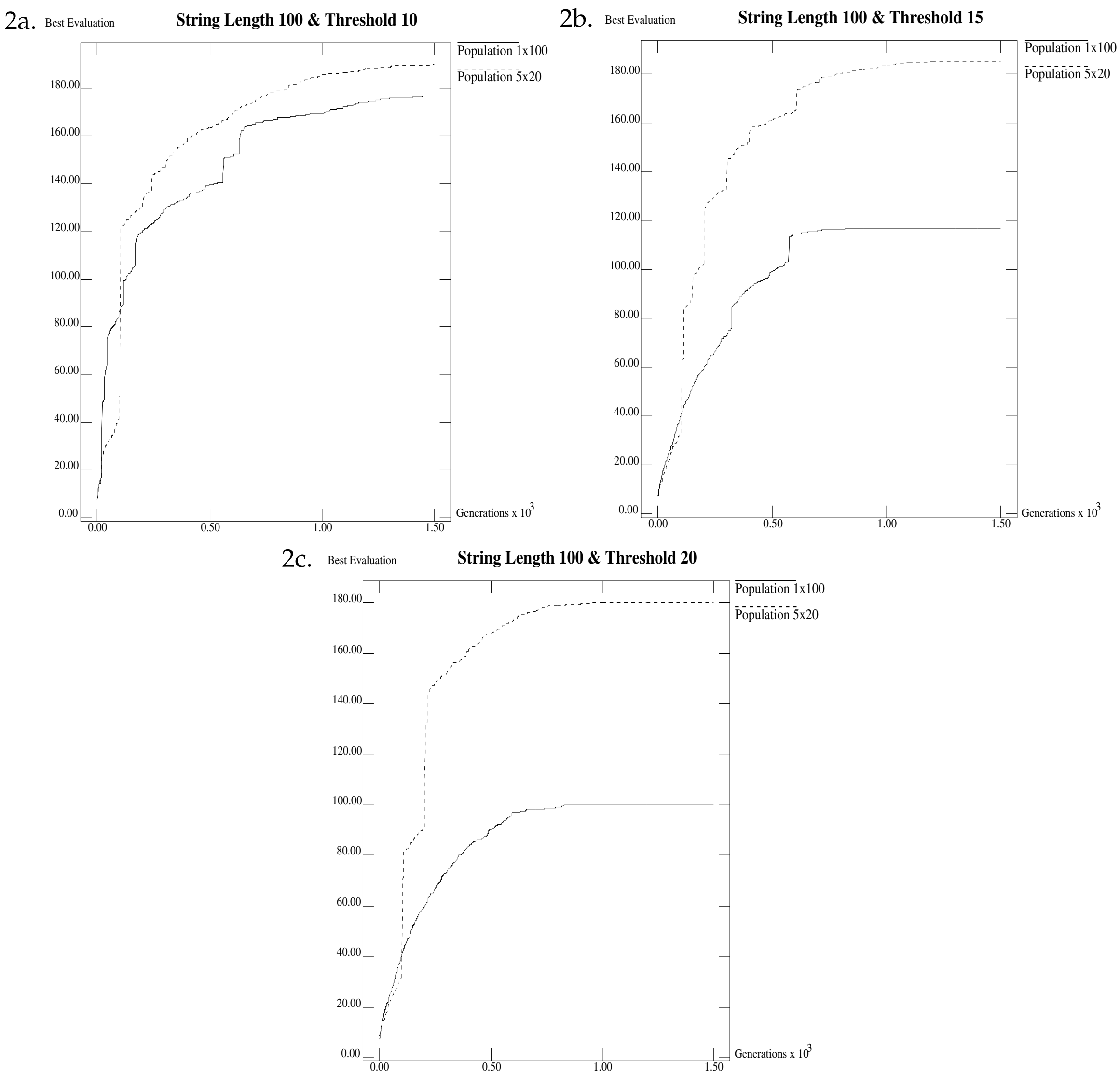


Figure 2: The failure of a simple GA to maintain multiple good points in the function space. This exemplifies the benefits of using divided subpopulations for explicitly preserving diversity. Average of 20 runs shown.

Admittedly, this problem is contrived to show the potential benefits of using parallel populations. The single large population converges to a suboptimal solution of either all ones or all zeros. With multiple subpopulations, each subpopulation can evolve to a different point. The eventual interaction between subpopulations can combine these points to find globally optimal solutions. In general, optimization problems may not have the same characteristics necessary to reveal such pronounced benefits of pGAs in comparison to traditional GAs. Nonetheless, pGAs have shown benefits in a large number of optimization problems, both in terms of the number of evaluations performed and the final results obtained [Husbands, 1990], [Whitley & Starkweather, 1990] [Davidor et al., 1993] [Gordon & Whitley, 1993].

The previous paragraphs showed that a simple genetic algorithm, with a single population, cannot maintain dissimilar points in the function space. The “Fundamental Theorem of Genetic Algorithms” does not address this. This theorem is stated below:

> “Short, low order, above average schemata receive exponentially increasing trials in subsequent generations” [Goldberg, 1989]

The inability to provide increasing trials to multiple dissimilar points is akin to the problem of genetic drift in biological systems. Due to stochastic sampling errors, a population will converge to a single point when presented with multiple points in the function space which have the same, or only an extremely small differential in their evaluations. In describing the effect of genetic drift, Goldberg and Richardson summarize this finding:

> "... 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].

This experiment was included to demonstrate that the simple genetic algorithm cannot maintain implicit parallelism in the latter portions of search. In systems which can maintain the parallelism in search, the problem is easily solvable. In the next section, the population is replaced by a single probability vector, which specifies the probability, in each position, of containing a specific value. A single vector can be used, because the parallelism (the ability to maintain multiple dissimilar high evaluation points in the function space) is not maintained in a single population GA. In modelling GAs with more than a single population, multiple probability vectors can be used. The use of multiple probability vectors is not examined here; however, it is returned to in the discussion of future directions.

2.2. Replacing the Population

The PBIL algorithm attempts to create a probability vector from which samples can be drawn to produce the next generation's population. As in standard GAs, it is assumed that the solution is encoded into a fixed length vector. Ignoring the contribution of mutation, the expected distribution of values in each position of the population during generation *G*, can be computed based upon the population of generation *G-1*. Assuming a fully generational GA (each member of the population is replaced in the subsequent generation), fitness proportional selection, and a general pair-wise recombination operator, the probability of value *j* appearing in position *i* in a solution vector *x*, in a population at generation *G*, can be computed as follows:

$$P(i,j) = P(x_i = j) = \frac{\sum_{v \in Population_{G-1} \wedge v_i = j} EvaluateVector(v)}{\sum_{v \in Population_{G-1}} EvaluateVector(v)}$$

This is simply a counting argument, weighted by the evaluation of each solution string. Given a population, a unique representation can be made by a probability matrix defined by the above equation. Two comments should be made about this representation. First, many populations will have identical probability matrices. Second, if the above representation was used to represent the population, and samples for generation *G* were drawn by sampling points based upon this distribution, the solution vectors produced are unlikely to improve over those in generation G-1. This is due to the implicit assumption that each bit position's value is independent of all other bit position's values across individual solution vectors. Traditional crossover does *not* assume this; a *pair-wise* crossover operator maintains more information between the bit positions, as the composition of the "children" solution strings is chosen from only two "parent" solution strings.

Although a population represented directly as mentioned above may not be useful for sampling in order to generate the next generation's vectors, a variation of the above representation can be useful. From a population of size N, consider probabilistically (based upon fitness) selecting N solution vectors for recombination. These newly generated vectors can be represented as a probability matrix by simply counting the number of occurrences of each value in each bit position. This method is described in greater detail by [Syswerda, 1992]. This probability matrix can be used to create a new population. This representation has been used to simulate crossover. It is compared to other crossover operators in [Eshelman & Schaffer, 1993].

In a manner similar to the methods described above, the population-based incremental learning (PBIL) algorithm uses a probability vector to describe the population of a genetic algorithm. In a binary encoded solutions string, the probability vector specifies the probability of each bit position containing a '1'. The probability of the bit position containing a '0' is obtained by subtracting the probability specified in the vector from 1.0. For example, see Figure 3. Although from this point, PBIL will be considered for solution vectors encoded in a binary alphabet, this method can be applied to higher-cardinality representations.

Population #1	**Population #2**	**Population #3**
0 0 1 1	1 0 1 0	1 0 1 0
1 1 0 0	1 1 0 0	0 1 0 1
1 1 0 0	1 1 0 0	1 0 1 0
0 0 1 1	1 1 0 0	0 1 0 1
Representation	**Representation**	**Representation**
0.5, 0.5, 0.5, 0.5	1.0 0.75 0.25 0.0	0.5 0.5 0.5 0.5

Figure 3: The probability representation of 3 small populations of 4 bit solution vectors; population size is 4. Notice that the first and third representation for the population are the same, although the solution vectors each represents are entirely different.

The PBIL algorithm attempts to create a probability vector which can be considered a prototype for high evaluation vectors for the function space being explored. A very basic observance of genetic algorithm behavior provides the fundamental guidelines for the performance of the PBIL. One of the key features in the *early* portions of genetic optimization is the parallelism in the search; many diverse points are represented in the population of a single generation. In representing the population of a GA in terms of a probability vector, the most diversity will be found in setting the probabilities of each bit position to 0.5. This specifies that generating a 0 or 1 in each bit position is completely random.

The PBIL algorithm uses the probability vector representation for defining a population. Rather than passively transforming each population into a probability vector, from which solution vectors are generated and recombined, etc., the aim of PBIL is to actively create a probability vector which, with high probability, represents a population of high evaluation solution vectors. Unlike the mechanisms inherent to a GA, operations are not defined on the population; rather, in PBIL, operations take place directly on the probability vector. These mechanisms are derived from those used in competitive learning.

In a manner similar to the training of a competitive learning network, the values in the probability vector are gradually shifted towards representing those in high evaluation vectors. A simple procedure to accomplish this is described below. The probability update rule, which is based upon the update rule of competitive learning, is shown below.

The probability update rule described above is the similar to the weight update rule in a competitive learning network when an output is moved towards a particular sample point. It is interesting to note that the each bit is examined independently. Above, it was claimed that a representing each bit independently dis-

$$probability_i = (probability_i \times (1.0 - LR)) + (LR \times vector_i)$$

$probability_i$ is the probability of generating a 1 in bit position *i*.
$vector_i$ is the *i*th position in the solution vector which the probability vector is moved towards.
LR is the learning rate

regarded much of the information which standard crossover preserved. The reason the assumption of independence is not detrimental, is that PBIL does **not** attempt to represent the entire population by the probability vector. Rather, PBIL represents a *single* point, based upon the vector with the highest evaluation, around which the next population of points should be generated. This is described in more detail below.

The step which remains to be defined is how to determine which solution vectors to move towards. If the good vectors were known *a priori*, the problem would, of course, already be solved. Several alternative methods have been explored. The basis of some of the earlier attempts involved generating a single vector, and deciding whether to push the probability vector towards the generated vector. However, a more effective method, which has proven empirically to be more resistant to getting caught in local minima, is to generate a number of vectors, all based upon the probabilities specified in the probability vector, and to push the probability vector towards the generated vector with the highest evaluation. After the probability vector is updated, a new set of vectors is produced based upon the updated probability vector, and the cycle is continued.

Generating a population of potential solutions, rather than a single solution vector, is an attempt to maintain the ability to explore large regions of space in a parallelized manner, as the GA does in the early stages of search. In the early stages of search, there is a large amount of diversity in the regions of the function space which are simultaneously explored. As the search progresses in PBIL, the values in the probability vector move away from 0.5, towards either 0.0 or 1.0. In the GA, this corresponds to the respective bit positions in the majority of the solution strings having the same value. As the search progresses, the population of the GA tends to converge around a good solution vector in the function space. Analogously to genetic search, the progression of search in PBIL converges around a single point. The PBIL algorithm and a standard GA face the same problem of premature convergence. In PBIL, as the probabilities become very close to either 0.0 or 1.0, the similarity in the vectors generated increases. One advantage which the PBIL algorithm offers is explicit control of how fast the population converges. The setting of the learning rate parameter, which can greatly effect the speed to convergence, is explored in Section 3.

A concern is that because only a single probability vector is used, it may have less expressive power than a full population GA. A GA which uses a population of points can represent a large number of points simultaneously. For example, in Figure 1, the representations for population #1 and population #3 are represented by probability vectors of 0.5; therefore, it is unlikely that the population members would be regenerated by sampling the probability vector. This appears to be a fundamental limitation of PBIL, as it is possible for a GA to contain either of these populations. However, for the reasons mentioned previously (genetic drift), in simulating genetic search, a traditional single population GA *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. Therefore, even if the members of the populations shown in Figure 3 had equally high evaluations, the GA would be unable to maintain them in its population, and would converge to only one. Similarly, in PBIL, the values of 0.5 will quickly be changed to favor either 0.0 or 1.0 through the search's progression; the probability vector can only represent one of the dissimilar points. Methods designed to address this problem, which implement explicit parallelism, are not explored here. However, it should be noted that many of the methods are applicable to both GAs and PBIL.

As the population which is represented by a probability vector is not unique, this aids in maintaining diversity in search. For example, members from population #1 and population #3 can be generated by the probability vector used to represent these populations. In traditional GAs, uniform crossover (see Appendix A) also often has the same characteristics from one generation to the next - extremely dissimilar children can be produced from the same parents.

As discussed previously, the GA does not only rely on crossover to perform extensive search. In the latter parts of search, mutation plays a prominent role. In an analogous manner, a similar operator is defined in PBIL. For PBIL, there are two ways of defining a mutation operator. The first is to perform the mutation directly on the vectors generated. The second method is to perform a mutation on the probability vector; this mutation can be defined as a small probability of perturbation on each of the positions in the probability vector. Both of these forms of mutation have the same effect as mutation in standard genetic algorithms: to help preserve diversity. For the experiments conducted in this study, the second form of mutation is implemented. The full algorithm is described in Figure 4; this is the original EGA algorithm as formulated in conjunction with [Juels, 1994]. The role of the crossover operator and the behavior of the population over multiple sequential crossover steps in the GA forms the basis of the EGA; the EGA is currently being analyzed from this perspective by [Juels, 1994].

To keep the analogy with a GA simple, the number of vectors generated at each iteration is defined to be the population size. The update rule to the probability vector is investigated from the perspective of competitive learning in the next section. The mutation operator and its benefits are explored more thoroughly in section 2.4.

2.3. The Probability Vector and Competitive Learning

The probability vector maintained by PBIL can be viewed as a prototype vector for generating solution vectors which have high evaluations with respect to the available knowledge of the function space. In each generation, the probability vector is adjusted to represent the current highest evaluation vector. As values in the bit positions become more consistent between the highest evaluation vectors produced in subsequent generations, the probabilities of generating the value in the bit position increases. As mentioned before, the unsupervised context of traditional competitive learning is no longer appropriate. The feature of interest is competitive learning's ability to find a prototype for high evaluation vectors. However, it should also be noted that the supervisory signals used in this algorithm are also non-standard. In most traditional supervised classification tasks, the sample points are pre-labeled. In PBIL, the class of high evaluation vectors is defined during the algorithm's search. In each population of points generated, the highest evaluation vector produced is defined to be in the class of interest. Finally, it should also be noted that the probability vector not only specifies the prototype based upon the high evaluations of the sample solutions, but also guides the search, which produces the next sample point from which to "learn".

The classes of problems to be addressed by PBIL are unlike many other problems for which competitive learning is often employed. In many domains to which CL is applied, one of the largest difficulties in training the CL-net is the lack of available training data. However, there is an abundance of training data in the class of problems attempted here. Training data is available through the evaluation of potential solution vectors. Nonetheless, to be efficient, the algorithm must minimize the number of function evaluations performed. Therefore, as information regarding the characterization of high evaluation vectors becomes available, it is incorporated into the search; the updated probability vector is used to generate the next population of sample points.

Although the method of defining classes is not common, the supervised training of this algorithm is similar to the learning vector quantization (LVQ) method presented by Kohonen [Kohonen, 1988] [Kohonen, 1989]. Kohonen's LVQ attempts to find the prototype vector of a set of known classes. In LVQ, the method by which the output unit is chosen is the same as in standard CL. However, if the winner is a class to which the point does not belong, the winner's weights are adjusted to move the prototype vector away from the misclassified point. As in standard CL, if the class is correct for the input point, the winner's weights are adjusted to more reliably classify the point. Similar extensions are possible for PBIL, such extensions include learning from misclassified examples. This is explored in greater detail in section 4.

The PBIL algorithm also has ties with early reinforcement learning algorithms [Moore, 1994], which use a probability vector approach to probabilistically chose an action. Such learning algorithms include the *linear reward- penalty algorithm* and its specialization, the *linear reward-inaction algorithm*. A good survey of early reinforcement algorithms can be found in [Kaelbling, 1990].

```
P ⟵ initialize probability vector. (Each position = 0.5)

loop # GENERATIONS

        # Generate Samples
        i ⟵ loop #SAMPLES
                sample_i ⟵ generate sample vector according to probabilities in P
                evaluation_i ⟵ evaluate (sample_i)

        # Find Best Sample
        max ⟵ find vector corresponding to maximum evaluation

        # Update Probability Vector
        l ⟵ loop #LENGTH
                P_l ⟵ P_l * (1.0 - LR) + max_l * (LR)

        # Mutate Probability Vector
        l ⟵ loop #LENGTH
                if (random (0,1] < MUT_PROBABILITY)
                        P_l ⟵ P_l * (1.0 - MUT_SHIFT) + random (0.0 or 1.0) * (MUT_SHIFT)
```

USER DEFINED CONSTANTS:

GENERATIONS: number of iterations to allow learning.
SAMPLES: the population size, number of samples to produce per generation
LENGTH: length of encoded solution
MUT_PROBABILITY: probability of mutation occurring in each position
MUT_SHIFT: amount for mutation to affect the probability vector
LR: learning rate

VARIABLES:

P: probability vector
$\text{sample}_{i..SAMPLES}$: solution vectors
$\text{evaluation}_{i..SAMPLES}$: evaluations of the solution vectors
max: solution vector corresponding to maximum evaluation

Figure 4: The basic PBIL algorithm. When using this algorithm, the highest evaluation vectors should also be maintained, and returned at the end of the procedure. This is the EGA algorithm as formulated with [Juels, 1994].

2.4. The Role of Mutation in GAs and PBIL

One of the goals of this paper is to examine the roles of the operators used in genetic algorithms to determine what their relevance is to the PBIL algorithm, and what their role should be. In this section, the role of the mutation operator is examined. The role of mutation is more pronounced in the latter stages of search, when diversity in the population is lost [Spears, 1992]. Crossover is valuable in the early portion of search as it takes large steps towards better solutions [Fogel, 1993]. In many optimization problems, much of the fine-tuning (moving from good regions of the function space to optimal solutions) in genetic search occurs because of mutation operator. The crossover operator is less effective, as described previously, as recombining similar chromosomes exclusively through the use of crossover does not introduce diversity or induce exploration of the function space. A good discussion of the trade-offs between mutation and crossover, and their roles in performing extensive search can be found in [Spears, 1992][Fogel, 1993].

This section is provided to show the importance of the mutation operator in genetic search and in PBIL. The role of mutation is examined in an easy sample problem, described below. The performance of a GA with and without mutation and PBIL with and without mutation is shown for the sample problem in Figure 5. Also included in Figure 5 is the performance of a GA which has mutation only in the later parts of the search, in particular, the mutation is turned on only after generation 200. These are included to provide an intuitive understanding the role mutations have in function space exploration. The performance of a next-step hillclimber is given; the details of the hillclimber can be found in Appendix B. For the problem attempted a solution vector is composed of 200 bits, and the population size for each algorithm is 100 vectors. The mutation rate for the genetic algorithm is 0.001. The mutations for the PBIL occur on the probability vector with probability 0.02. In each mutation, the probability vector is shifted by a value of 0.05 in a randomly chosen direction. The graph shown represents the average performance of 20 runs, allowed to perform 50,000 function evaluations each (500 generations). Each of the 20 variables x_0 - x_{19} are encoded in base 2, with 10 bits.

$$f(\vec{x}) = \prod_{i=0}^{19} (1.5 - |(0.023 \times i) - x_i|)$$

$$0.0 \le x_0 \ldots x_{19} < 1.024$$

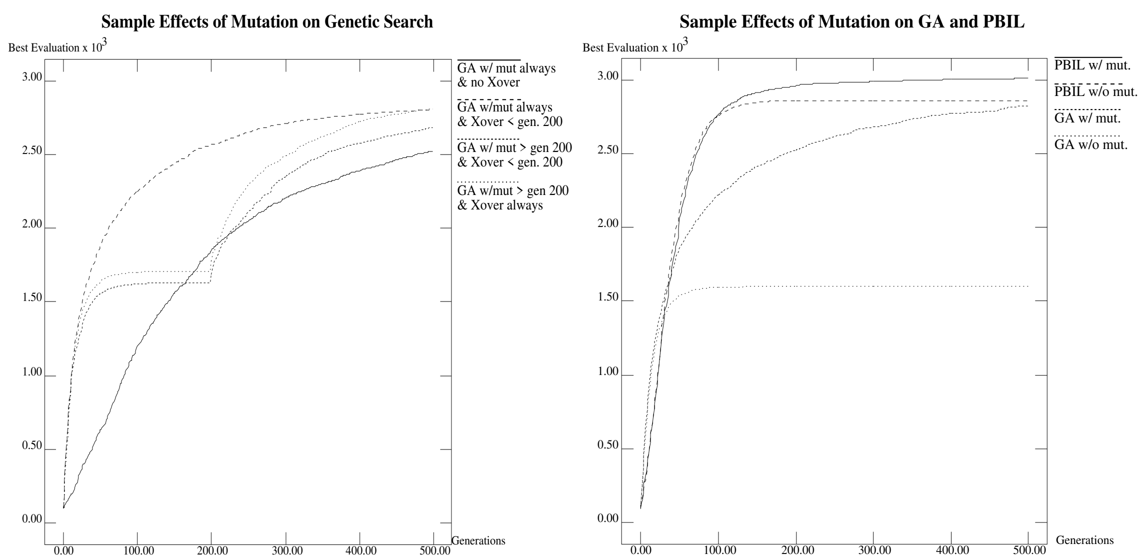


Figure 5: The effects of mutation on the search ability of a standard genetic algorithm and PBIL. The left graph shows GA runs with mutation and crossover operators on before and after generation 200 (see diagram). In the GA runs in the right graph, the crossover operator is constantly on in both runs, and the mutation operator is on only in the first run. The highest evaluation the hillclimber (Appendix B) was able to achieve was 2969, after 50,000x20 evaluations (758 restarts).

Although this is not a hard function to optimize, part of the difficulty in finding optimal solutions for this problem is the simple binary encoding used. The difficulty exists because of "hamming cliffs" present in the encoding. These are points at which the representation of the next sequential number is largely dissimilar to the current number's representation. For example, consider the binary representation for 255 and 256: 011111111 and 100000000. This type of encoding increases the difficulty of search. The problem can be overcome through the use of gray-coding. Gray coding ensures that the representations of subsequent numbers differ only by a single bit. Gray code is not used for this problem to display one of the many potential difficulties in creating arbitrary black-box function optimization techniques.

Although it is clear from the above example that mutation can aid PBIL in finding better final solutions, mutation may not play as crucial a role in PBIL as it does in standard genetic search. The manner in which potential solutions are probabilistically created from the prototype vector allows sampling a diverse set of points. The role of mutation is to prevent the prototype vector from too quickly converging to an extreme value (either 0.0 or 1.0) in each of the bit positions.

3. Examining the Effects of Changing the Learning Rate

There are four parameters which can be adjusted in the PBIL algorithm described to this point: the population size, the learning rate, the mutation probability and the mutation shift (the magnitude of the effect of a mutation on the probability vector). In consideration of the length of the paper, this section will concentrate only on the setting of the learning rate parameter, as this parameter has no counterpart in traditional GAs. Although the settings of population size and the mutation probability and shift can have a significant impact on the effectiveness of PBIL and a GA, there is an abundance of literature describing the tuning of the analogous parameters in standard genetic algorithms. Although extensive testing has not been conducted with different population sizes and different mutation rates with PBIL, it is suspected that much of this literature should yield valuable insights into appropriate setting of the parameters in PBIL. Some references for the setting of population size are [Muhlenbien, 1990] [Goldberg,1989] [Goldberg, 1992]. References in which settings of mutation rates are discussed include [De Jong, 1975], [Baeck, 1993], [Whitley and Starkweather, 1990].

The learning rate parameter has a larger effect in PBIL than in standard competitive learning. As with competitive learning, the learning rate affects how fast the prototype vector is shifted to resemble a correctly classified point. Since in PBIL, the probability vector is used to generate the next set of sample points, the learning rate also affects which portions of the function space will be explored. The setting of the learning rate has a direct impact on the trade-off between exploration of the function space and exploitation of the exploration already conducted. In this context, exploration is the ability of the algorithm to search the function space thoroughly, while exploitation refers to the algorithm's ability to use the information it has gained about the function space to narrow its future search. For example, if the learning rate is 0, there is no exploitation of the information gained through search. As the learning rate is increased, the amount of exploitation increases, and the ability to search large portions of the function space diminishes. This is illustrated in the following example.

Eshelman and Schaffer used the following problem to compare how much exploration is done by various crossover operators [Eschelman and Schaffer, 1993]. In a 100 bit solution vector $\bar{\mathbf{x}}$, the evaluation of the vector is defined as follows:

$$f(\vec{x}) = MAX(o(\vec{x}), z(\vec{x}))$$

where:
$o(\bar{x})$ is the number of ones occurring in $\bar{x}$,
$z(\bar{x})$ is the number of zeros occurring in $\bar{x}$.
MAX returns the larger of its two arguments.

This is similar to the first problem described in the paper, although this does not impose the requirements of building consecutive ones from the left, and consecutive zeros from the right. Eschelman and Schaffer attempted to determine how soon the commitment to either all zeros or all ones was made. In other words, if the final solution string consisted of all zeros, what was the largest number of ones ever produced in a solution string at any time during the run, and vice-versa.

The higher the learning rate parameter is set, the faster the algorithm will focus search. The lower the learning rate, the more exploration will occur. In the Figure 6a, four learning rates are explored which show rapid convergence. In Figure 6b, the learning rates are greatly lowered, and the rate of convergence is significantly slower. In Figure 6c, the convergence of the population for a simple genetic algorithm is shown. The GA run is shown both with and without elitist selection. For a more complete description of elitist selection, see Appendix C.

From the figures, it is clear that for some values of the learning rate in PBIL, a simple GA exhibits a greater ability to perform extensive exploration. For these settings, it is possible to design problems in which the GA can be ensured to be perform better than PBIL on average. The types of problems may be as simple as a time-varying function which gives a reward for finding 40 ones, after performing some large number of evaluations (in this case 10000 would make the GA (w/o elitist selection) more successful than the PBIL algorithms shown in Figure 6a). However, lowering the learning rate in PBIL can address some of these problems. It should be noted, however, that if the learning rate is set too low, it is possible to cripple the algorithm so that it may take an enormous number of samples before it is able to shift the probability vector enough to escape the initial oscillations, as is shown in Figure 6b.

The implications of this experiment are important to other optimization problems than the one shown. If the learning rate is high, the initial populations generated will largely determine the focus of the search, without enabling the algorithm to explore the function space. If the function space does not contain local optima, a high learning rate may work well. However, if local minima could be a problem, lower learning rates allow greater exploration.

4. Extensions to the Basic Algorithm

As noted earlier, some of the mechanisms used in PBIL are similar to those used in Kohonen's version of vector quantization. The greatest difference in Kohonen's LVQ and PBIL is the target outcome of each method. Kohonen's LVQ attempts to create decision boundaries for classes based upon known labeled examples. PBIL attempts to optimize a function based upon labeled examples. Regardless of these differences, the procedures used are very similar. In Kohonen's version of vector quantization, the learning is supervised as the classes for each of the sample points is known *a priori* [Kohonen, 1988]. Similarly, the class of interest is already defined in the PBIL procedure. In Kohonen's LVQ, the network is trained with both positive and negative examples. In particular, if the classification made by the network is correct, the weights of the winning output are strengthened; if the classification is incorrect, the weights are weakened. In an analogous manner, it is possible to modify PBIL to train the prototype vector with more than the single best solution vector. The remainder of this section is devoted to discussing two methods of using more than the single best generated vector.

In each iteration of PBIL, a number of vectors are generated. However, in the version of the algorithm described to this point, the prototype vector is only adjusted based upon the single best solution vector generated in the current generation. When large populations are used, this has the potential of ignoring a large amount of the work and exploration performed by the algorithm. Several alternatives are possible. The first is to move the probability vector in the direction of the best **M** vectors, where **M** << **N** (**N** is the number of vectors generated). This can be realized through several approaches. The first is the straightforward implementation: the prototype vector is moved equally in the direction of each of the selected vectors. An alternative method is to move the probability vector only in the positions in which there is a

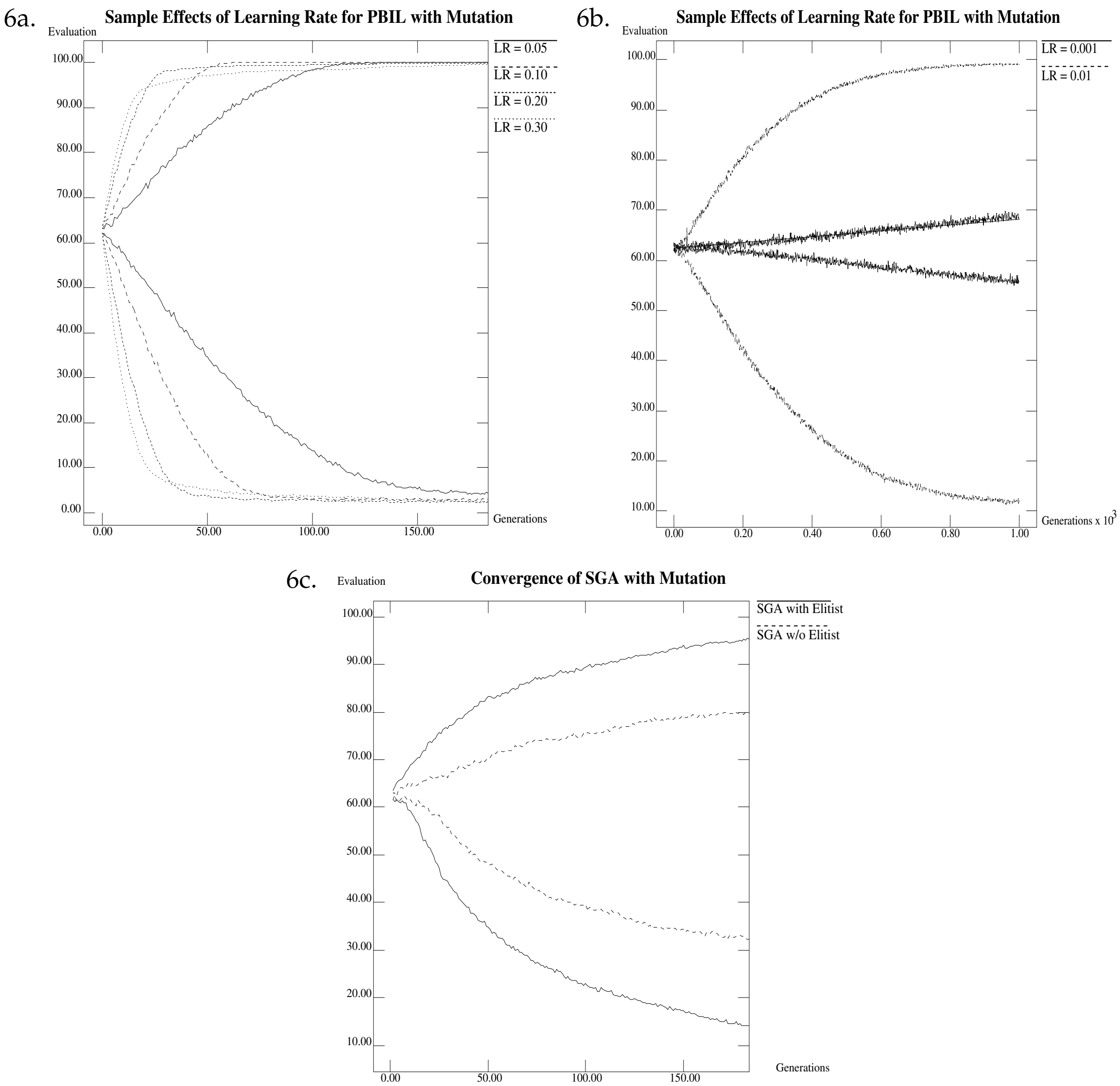


Figure 6: The effects of learning rate on PBIL's ability to search the function space. Each line has two parts, one moving towards the top of the graph, the other to the bottom. The line moving towards the top of the graph represents the maximum evaluation in the generation. If the best solution in the generation is found because of the number of 1s, then the line extending toward the lower regions represents the maximum number of 0s in each generation. If the best solution if found because of the number of 0s then the lower line represents the maximum number of 1s. Average of 20 runs shown.

consensus in all, or most, of the **M** solution vectors. For the positions in which there is no consensus in the M generated vectors, the prototype vector is not shifted. Another rule could base the move of the prototype vector on the relative evaluations of the best M vectors. However, to this point, the algorithm can ignore issues regarding scaling, as only the rank is important. As described earlier in the paper, problems regarding scaling have often proved to be detrimental in GA search. Although methods which require such scaling may prove to be useful, this tactic is not explored in this paper.

An alternative to only using multiple high evaluation vectors is to use low evaluation vectors. In this implementation, the single highest evaluation vector has the same role as in the original implementation.

However, the prototype vector is also moved away from the lowest evaluation vector. A naive implementation of this is to simply move towards the complement of the lowest evaluation vector. However, this method will not work well in the advanced stages of search since the best and worst vectors may be similar. As the prototype vector becomes more fixed towards either 0.0 or 1.0 for each bit position, the probability of generating very similar vectors increases; therefore, the difference between the best and worst generated vectors will diminish as the search progresses (of course, this does not imply that their evaluation will not be vastly different). If the bit-wise difference is small, moving away from the worst vector could be counter-productive as it will also move away from the best vector in the majority of the bit positions. A simple modification, which has proven to work well, is to only move away from the bits in the worst vector which are different than those in the best vector.

One of the drawbacks with introducing either of the above two extensions to PBIL, is that both introduce more parameters to the algorithm. For example, the first method, moving towards multiple good vectors, requires the number of high evaluation vectors considered, **M**, to be defined. The second method, moving away from bad vectors, requires the specification of the learning rate for the negative examples. In the next section, standard PBIL and PBIL with moves away from negative examples are examined empirically. Several settings for the negative learning rate are investigated.

5. Empirical Analysis

In this section standard PBIL, PBIL which includes learning from negative examples, and a simple GA are compared empirically. All PBIL algorithms have the following parameters (unless otherwise noted): learning rate 0.1, mutation probability 0.02, mutation shift 0.05, and population size = 100. The algorithms compared are:

1. standard PBIL - Negative Learning Rate = 0.0 (S-PBIL). This is the EGA algorithm as developed in conjunction with [Juels, 1994].

2. PBIL - Negative Learning Rate = 0.025 (PBIL2 0.025)

3. PBIL - Negative Learning Rate = 0.075 (PBIL2 0.075)

4. PBIL - Negative Learning Rate = 0.1 (PBIL2 0.1)

5. PBIL - Naive Implementation of Learning from negative examples, Negative Learning Rate = 0.025 (PBIL-N) For details, see section 4.

6. PBIL with only move away from low evaluation vectors, do not move towards best. Learning Rate = 0.0; Negative Learning Rate = 0.1 (PBIL-LOW)

7. Standard Genetic Algorithm; Population Size = 100; Mutation rate = 0.001; Crossover = 2pt; Generational Model with Elitist Selection. (SGA) See Appendix C for a more detailed explanation.

8. Multiple Restart, Next-Step Hill Climber. See Appendix B for a detailed explanation of the algorithm. This algorithm was allowed 100x2000 evaluations per run, to match the number of evaluations per run in the other algorithms. The evaluations presented here are highest evaluations per run, averaged over 20 runs. Note that in all of these problems, the hillclimber restarted itself in random positions a minimum of 81 times (bin pack 1), and a maximum of 2397 times (De Jong's F2).

Several different values of the negative learning rate are included as it has very little or no other literature devoted to its setting. Although it is not expected that PBIL-N or PBIL-LOW will be able to optimize the harder functions as well, they are included for a base-line reference.

The twelve problems on which each of the algorithms are compared can be divided into five groups. The first three are known NP- complete problems: Jobshop Scheduling, Traveling Salesman Problems, and Bin Packing problems. The fourth set of problems is often used in genetic algorithm literature to gauge the performance of genetic algorithms; they are numerical optimization problems. The fifth category of problems can best be described as those which are designed to trap GAs in local optima or plateaus. Two problems are explored in this category: the first is the order-4 deceptive problem, a problem which is extremely difficulty for many single population GAs to optimize, the second is a checkerboard problem, as suggested by [Boyan, 1993]. The remainder of this section describes the problems, their encoding, and the performance of the seven algorithms on each of the problems.

In this section, the performance of each algorithm is shown in a graph representing the best evaluation in each generation, averaged over 20 runs. With the description of each of these experiments, the performance of a multi-start next-step hillclimber, described in detail in Appendix B, is also provided.

5.1. Jobshop Scheduling Problems

The job-shop scheduling problem is an important problem which is among the worst members of the NP-complete problems [Garey & Johnson, 1979]. Recently, genetic algorithms have been applied to the problem, as it is difficult for conventional search based methods to find near-optimal solutions in a reasonable amount of time [Fang et. al, 1993].

The following description of the job shop problem is given in [Fang et. al, 1993]:

> "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 the following manner: a string of j * m integers are encoded into a binary bit string, **x**. The integers are in the range of [1...j]; therefore, there are $\log_2 j$ * j * m integers in the encoding. A circular list, **C** of jobs is maintained. **C** is initialized to 1...j. The following procedure is repeated until C is empty:

1. read the next $\log_2 j$ bits from **x**
2. **i** $\longleftarrow$ convert the bits to an integer
3. **job y** $\longleftarrow$ find the **i**th entry in list **C**
4. **task t** $\longleftarrow$ find the next task to be scheduled for **job y** (this specifies the machine needed)
5. find the first viable time slot for **task t**, subject to the constraints that it must be
    - after the last scheduled task for **job y**
    - in a space which is at least as long as is required **task t**

6. increment the last scheduled task counter for **job y**. If **job y** is done, remove it from **C**.

The encoding here used a bit string to encode the integers in the range of 1...j. Fang et. al used chunks which are atomic for the GA [Fang et. al, 1993]. Although the encoding used here is certainly not the easiest space in which to find solutions, 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 Figure 7 and Figure 8.

The hillclimbing algorithm was unable to perform as well as the other algorithms on this problem, as it achieved evaluation of 0.00097 and 0.00077 on the 10x10 and 20x5 jobshop scheduling problems, respectively.

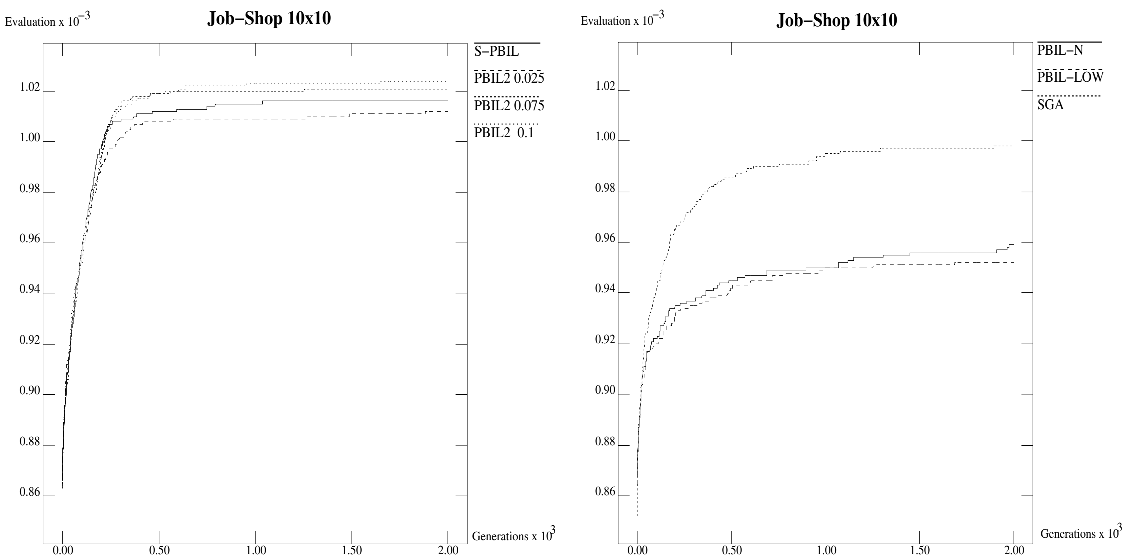
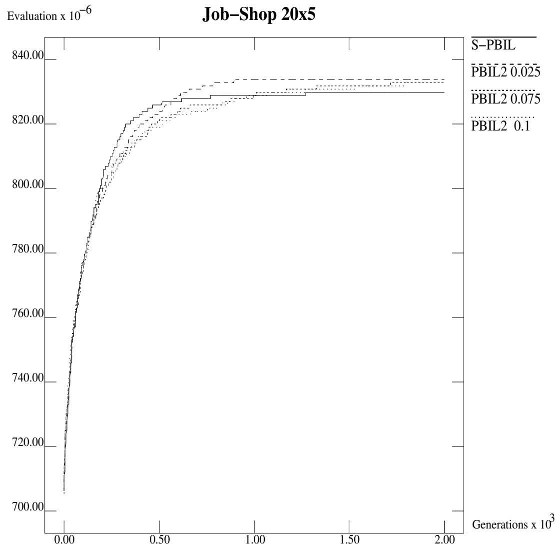


Figure 7: Job-Shop, 10x10



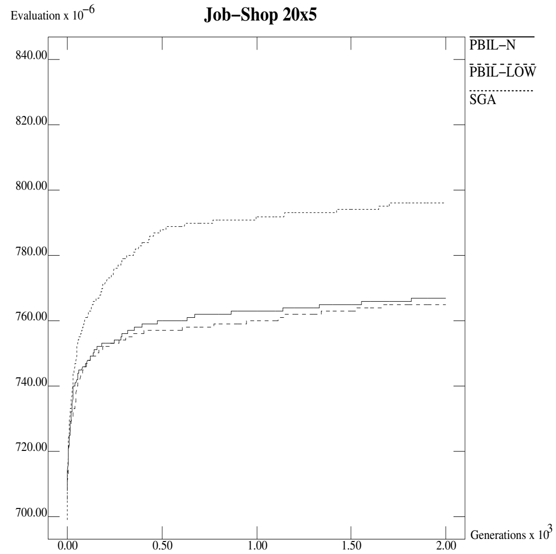


Figure 8: Job-Shop, 20x5

5.2. Traveling Salesman Problems

The Traveling Salesman Problem (TSP) is perhaps the most famous of the NP-complete problems. A version of the TSP is examined here in which the distances between cities are symmetric, and each of the cities lies on a two dimensional plane. However, it should be noted that these assumptions are not used explicitly by the algorithms tested. The object of the problem is to find the shortest length tour which visits each city exactly once, and returns to the original city.

For a problem with **N** cities, the encoding requires a bit string of size $\mathbf{N}\log_2\mathbf{N}$ bits. A city is assigned a contiguous substring of length **M** ($\mathbf{M} = \log_2\mathbf{N}$). Each substring is interpreted as a binary integer in the range of [0, N-1]. The city with the lowest integer value comes first in the tour, the city with the second 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, although not very effective, is commonly used when encoding the tour in a binary string. Again, it is used here to produce results which can be compared to other results given in literature. As the length of the tour is to be minimized, the evaluation of the potential solution is 1/Tour_Length. Two small problems were attempted: the standard 30 city TSP, and the 50 city TSP [Whitley et al., 1989]. The results are shown in Figure 9 and Figure 10.

The hillclimbing algorithm was unable to perform as well as the other algorithms on this problem, as it achieved evaluations of 0.00194 and 0.00155 on the 30 and 50 city TSP problems, respectively.

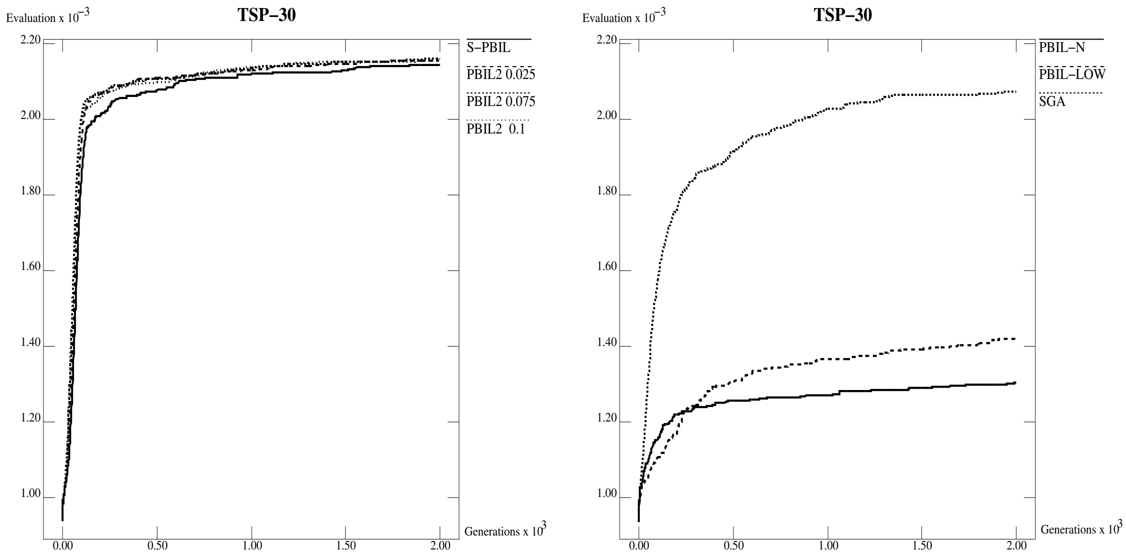


Figure 9: TSP 30 City.

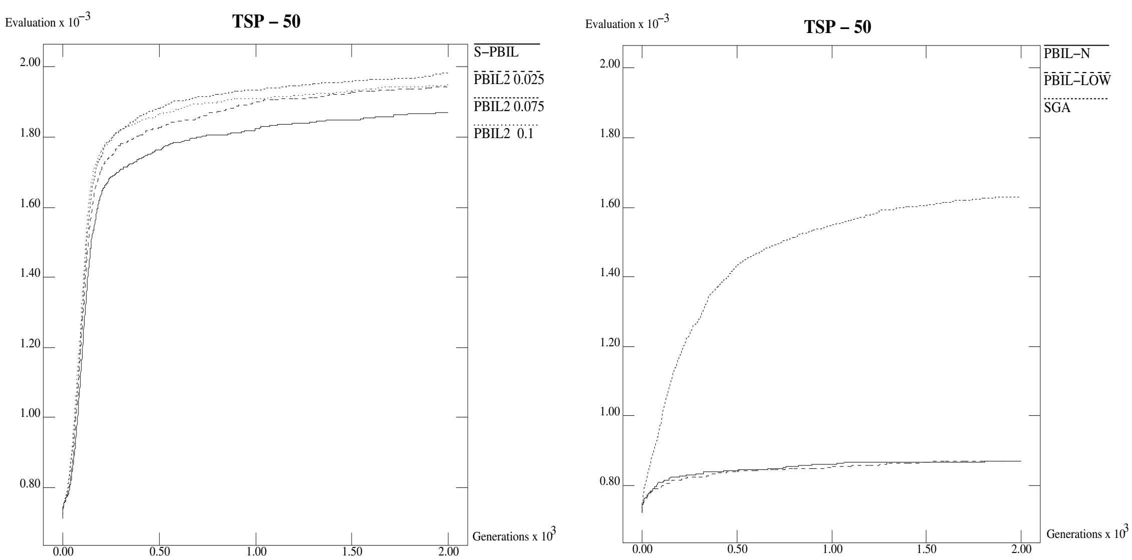


Figure 10: TSP, 50 City.

5.3. Bin Packing

In this problem, there are **N** bins of varying capacities and **M** elements of varying sizes, between 0.0 and 1.0. The problem is to pack the bins with elements as tightly as possible, without exceeding the maximum capacity of any bin. This problem is also known to be NP-complete [Garey and Johnson, 1979].

In the problem attempted here, the error of a particular solution is measured by:

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

where:
CAP_i is the capacity of bin i
$ASSIGNED_i$ is the total size of the elements assigned to bin i.

The problem's 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**. The substring is converted from binary to decimal. The value revealed is the bin in which the element is placed. As the error in packaging, **ERROR**, is to be minimized, the evaluation of the potential solution is 1.0 /**ERROR**.

Three problems are attempted, each with a string size length of 512 bits. The problems attempted are of the following sizes: 16 bins, 128 elements (Figure 11); 4 bins, 256 elements (Figure 12), 2 bins, 512 elements (Figure 13).

It is interesting to note that on the second bin packing problem, the hillclimbing procedure was consistently able to outperform all of the other algorithms (average evaluation was 168,078). On the first bin packing problem, the hillclimbing algorithm did not perform well (average evaluation 26). On the third bin packing problem, it was able to perform well, but not the best overall (average evaluation 28,677,215).

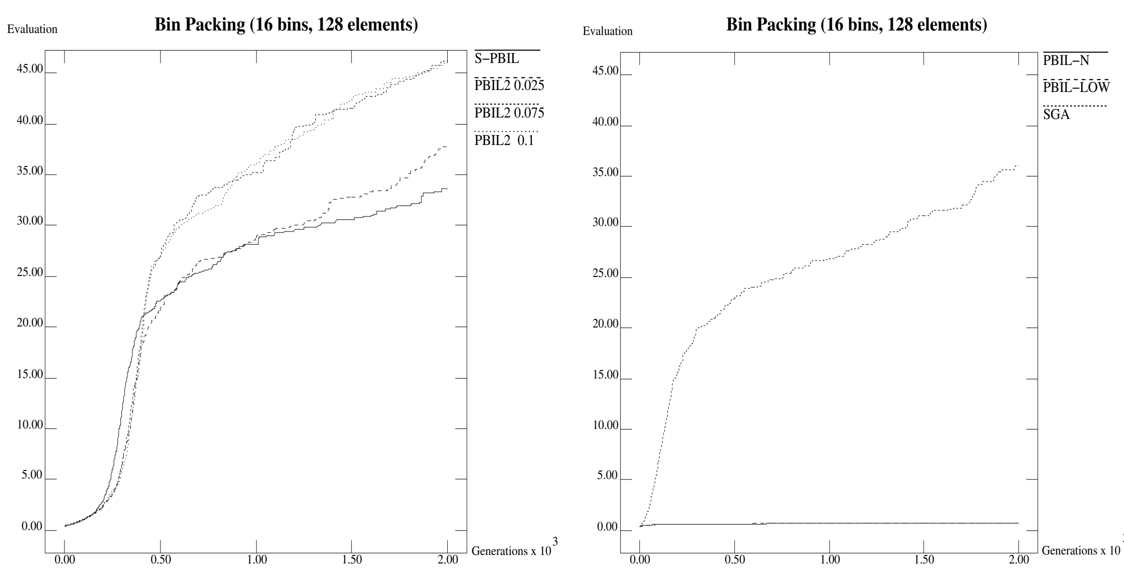


Figure 11: Bin Packing (16 Bins, 128 Elements)

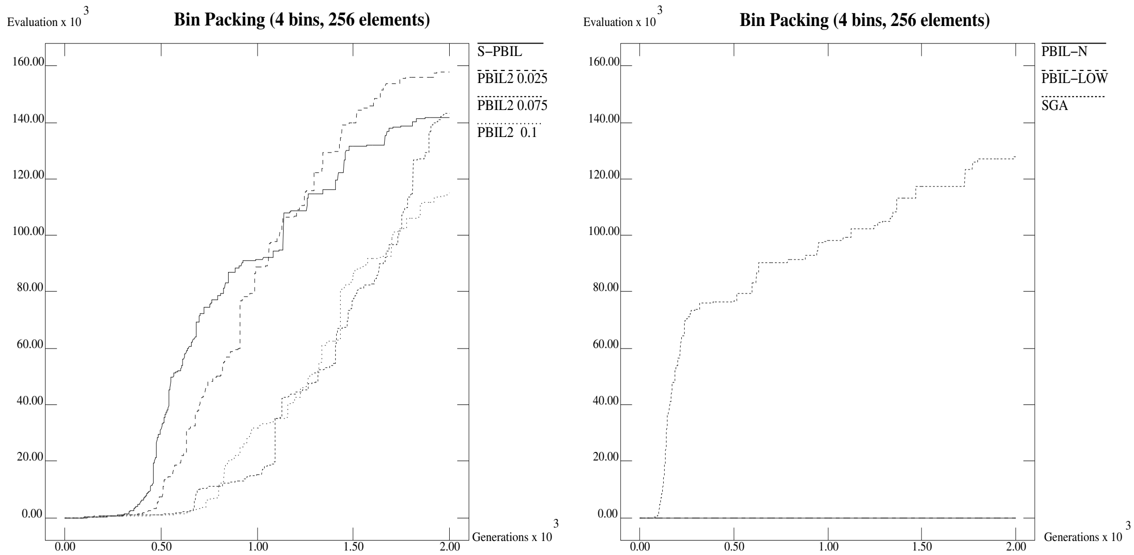


Figure 12: Bin Packing (4 Bins, 256 Elements)

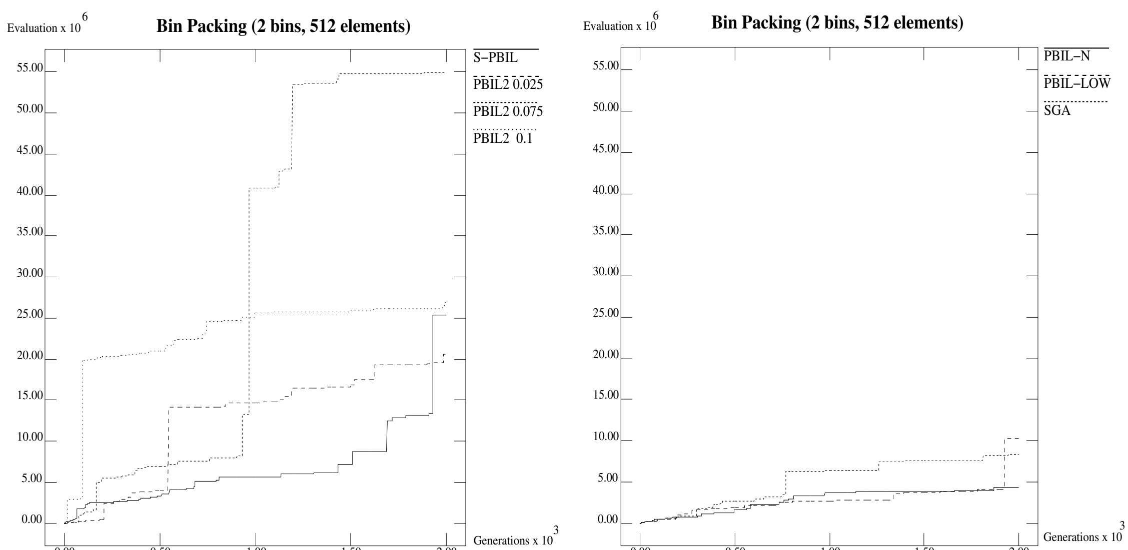


Figure 13: Bin Packing (2 Bins, 512 Elements)

5.4. Standard Numerical Optimization Problems

In addition to NP-complete problems, PBIL's ability to optimize functions which are common to genetic algorithm literature is also examined. The first two functions examined, De Jong's F2 and F3 [De Jong, 1975] are easily optimized functions on which both PBIL and the standard genetic algorithm do very well. These functions are rather small, and are easy to optimize for all of the methods examined here. Davis has explored the value of using these functions as standard benchmarks in [Davis, 1991]. The third function is based upon a function found in the paper by Gordon & Whitley; it in turn are based on a functions by Griewangk [Gordon & Whitley, 1993]. The encoding of the functions are slightly modified from their original encodings; the problems and the modifications are described below.

Each function attempted here used standard binary encoding for all of the variables. Each variable was encoded as a contiguous string in the solution vector. The problems are optimized as maximization problems.

5.4.1. De Jong F2

The only difference between the encoding of De Jong's F2 used here and in its original formulation is that it was originally formulated as a minimization problem, with the absolute minimum at 0.0. In this formulation, the problem is a maximization problem. The evaluation of the original F2 is subtracted from the maximum of the function, 3905.93. The function is shown below. The results are shown in Figure 14.

All of the algorithms performed well on this problem. The hillclimbing algorithm's average evaluation was 3905.93 (the optimal).

$$f(\vec{x}) = (3905.93) - (100) \times (x_1^2 - x_2)^2 + (1 - x_1)^2$$

$$-2.048 \le x_1 \ldots x_2 < 2.048$$

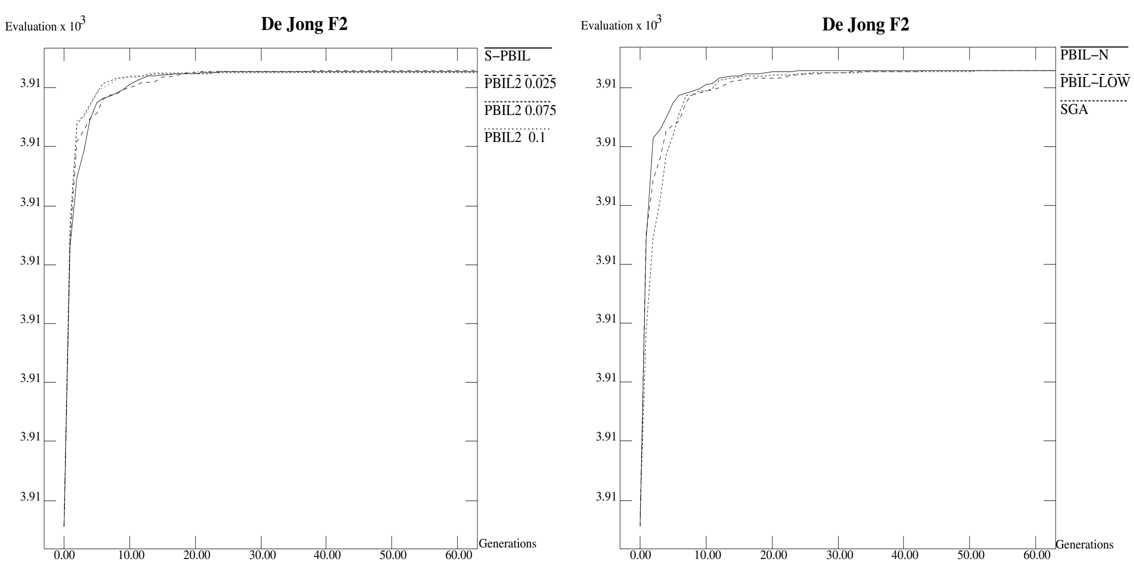


Figure 14: De Jong's F2.

5.4.2. De Jong F3

As with the previous problem, De Jong's F2, the only difference between the encoding of De Jong's F3 used here and in its original formulation is that it was originally formulated as a minimization problem. In this formulation, the problem is a maximization problem. A constant value of 30.0 is added to the function to make the global minimum at zero. The evaluation of the new F2 is subtracted from the maximum of the function, 55.0, to make the problem a maximization problem with a minimum of 0.0. The resulting function is shown below (in non-simplified form). The results are shown in Figure 15.

All of the algorithms performed well on this problem. The hillclimbing algorithm's average evaluation was 55.0 (the optimal).

$$f(\vec{x}) = 55.0 - \left(\left(\sum_{i=1}^{5} [x_i] \right) + 30.0 \right)$$

$$-5.12 \le x_1 \ldots x_5 < 5.12$$

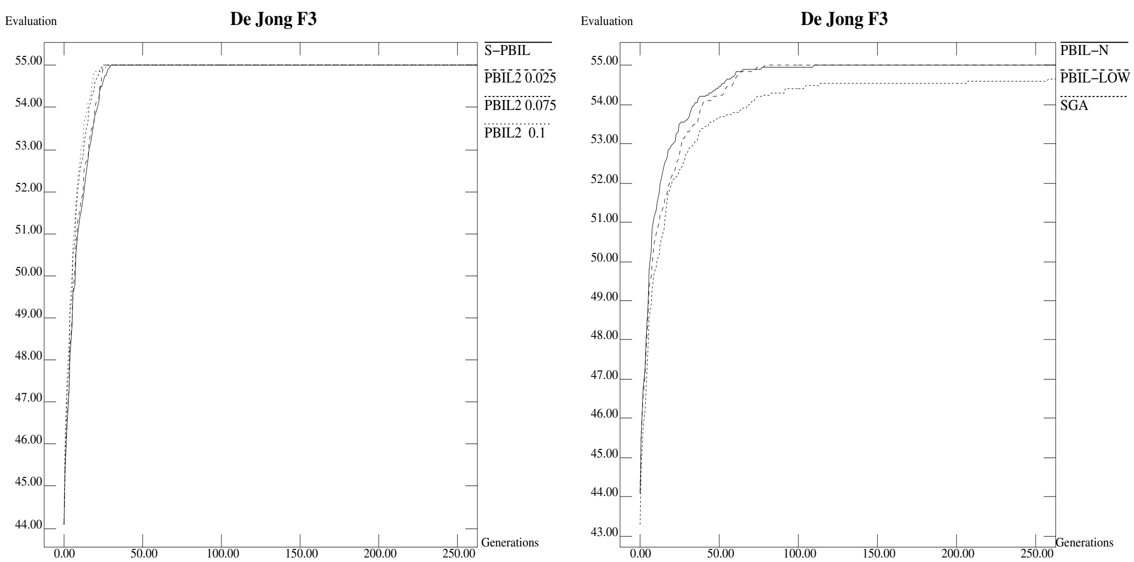


Figure 15: De Jong's F3.

5.4.3. Griewangk

This problem was taken from [Gordon & Whitley, 1993]. The problem is originally attributed to Griewangk. The original problem was a minimization problem. In order to cast the problem as a maximization problem, the problem is to maximize the reciprocal of the original function. To ensure that a value of 0.0 does not appear in the denominator, 0.1 is added in the denominator. The function is shown below. The results are shown in Figure 16.

$$f(\vec{x}) = \frac{1.0}{0.1 + \left(\sum_{i=1}^{10} \frac{x_i^2}{4000} - \prod_{i=1}^{10} \cos\left(\frac{x_i}{\sqrt{i}} \right) + 1 \right)}$$

$$-512 \le x_1 \ldots x_{10} < 512$$

The Hillclimbing algorithm was able to achieve an average final score of 10.6223. This is the highest evaluation achieved on the this problem with any algorithm tested in this study.

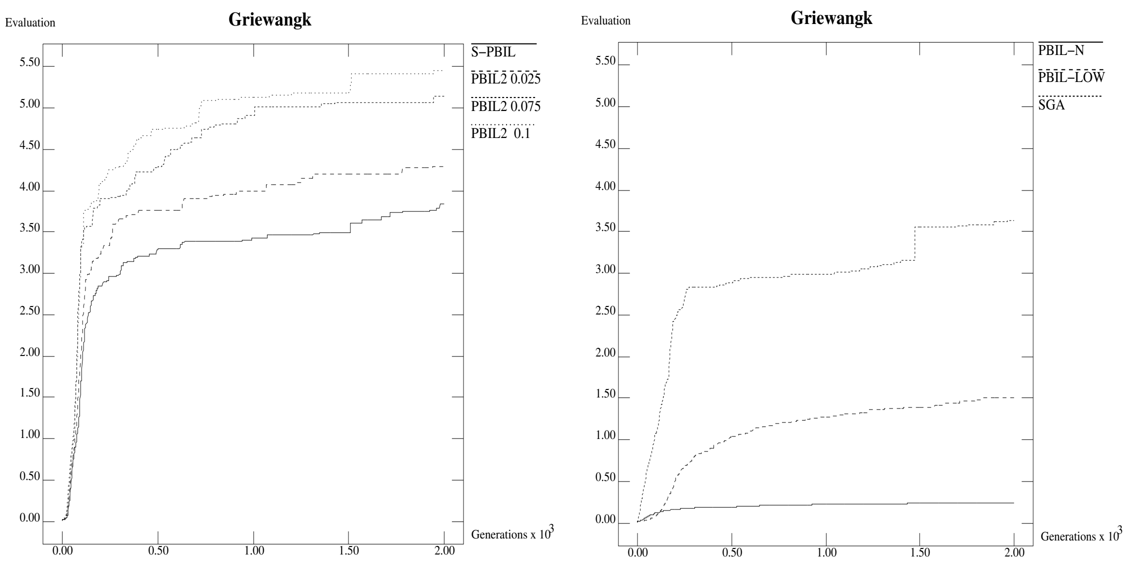


Figure 16: Function Based Upon Griewangk's function [Gordon & Whitley, 1993].

5.5. Strong Local Optima Problems

The last two problems studied in this paper are the order-4 deceptive problem and the checkerboard problem. The order-4 deceptive problem was originally suggested by Whitley & Starkweather [Whitley & Starkweather, 1990]. This problem is composed of 10 sub-problems, each of which is composed of 4 bits. The bits of each sub-problem are maximally distributed through the 40 bit vector. The evaluation of each 4 bit problem is show below. The objective is to maximize the sum of the evaluations. This is a vary hard problem for GAs to optimize as the vector of all 0's has an extremely large basin of attraction, while the vector of all 1's - the optimal solution - has none. However, this problem should be less deceptive for methods of optimization which very quickly restart search in random locations.

5.5.1. Order-4 Deceptive

Each of the 10 subproblems is evaluated as shown in Table I, each bit string is maximally dispersed throughout the entire solution string. The results are shown in Figure 17.

The average evaluation by the hill climbing algorithm was 291.40, this is higher than any of the algorithms tested. There were approximately 1,458 restarts in the 2000x100 evaluations performed during each run. The average evaluation of each search, before restarting, was 282.48.

Table I: Evaluations for Order-4 Fully Deceptive Problem

SOLUTION STRING	EVALUATION	SOLUTION STRING	EVALUATION
1111	30	0110	14
0000	28	1001	12
0001	26	1010	10
0010	24	1100	08
0100	22	1110	06
1000	20	1101	04
0011	18	1011	02
0101	16	0111	00

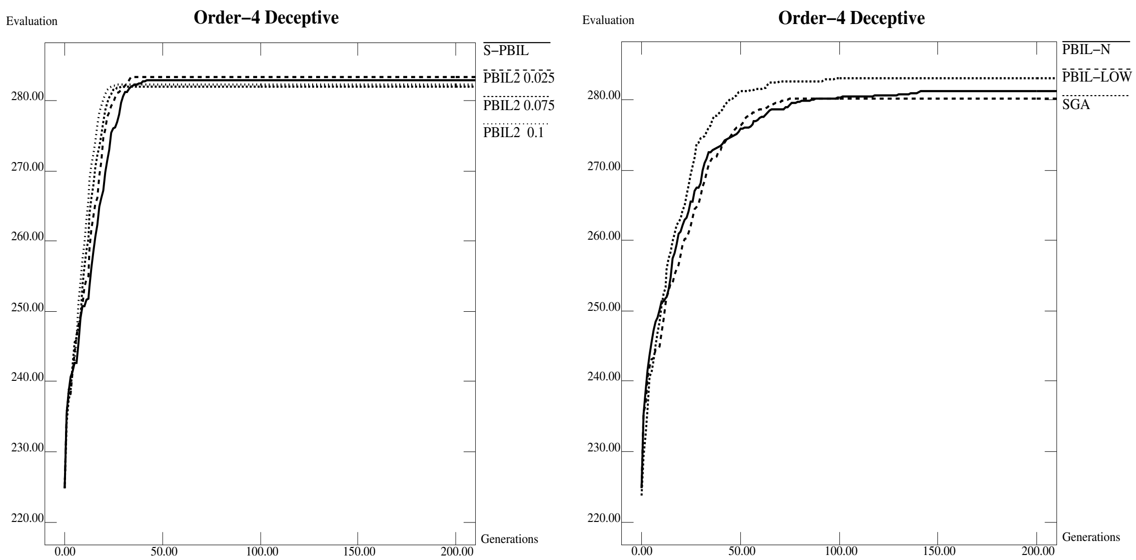


Figure 17: Tenfold Order 4 Deceptive Problems - Each subproblem is maximally interleaved.

5.5.2. Checkerboard Problem

This problem was originally suggested by [Boyan, 1993]. This problem requires a 400 bit solution string, which is interpreted as a 20x20 grid. The objective of the problem is to create a checkerboard pattern of 0's and 1's on the 20x20 grid. Each location with a value of 1 should be surrounded in all four directions by a value of 0, and vice-versa. Only the primary four direction are considered in the evaluation. The evaluation is measured by counting the number of correct surrounding bits for the *present* value of each bit position for a 18x18 grid, centered in the 20x20 grid. In this manner, the corners are not counted in the evaluation. There is no need to interpret the grid as a toroid as only an 18x18 grid is used. The maximum evaluation is 1296 (18x18x4). In addition to the methods compared in the previous problems, a parallel GA with 10 populations and 10 members per population was also tested. The results are shown in Figure 18. The hill climber's average evaluation was 1148.

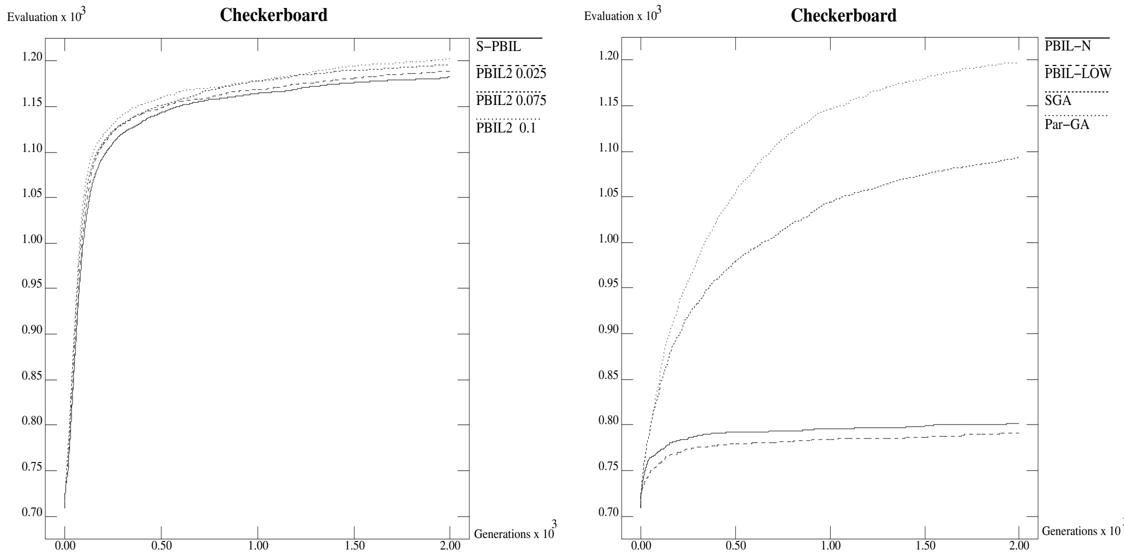


Figure 18: The Checkerboard Problem.

5.6. Summary of Empirical Results

A summary of the results in this section is presented in Table II; this table shows the problems on which each method performed the best. Note that all of the PBIL algorithms which used a non-zero positive and negative learning rate are grouped together. In the cases in which all algorithms achieved nearly identical performance (De Jong's F2 and F3), the best algorithm was chosen to be the one which found the solution in the fewest function evaluations.

Table II: Summary of Results

Algorithm	Number of "Wins"	Problems in which "Wins" occurred.
S-PBIL	0	N/A
PBIL2 (0.025, 0.075, 0.1)	8	Jobshop 10x10, Jobshop 20x5, TSP 30, TSP 50, Bin Pack 1, Bin Pack 3, De Jong's F3, Checkerboard
PBIL-N	0	N/A
PBIL-LOW	1	De Jong's F2
Standard Genetic Algorithm	0	N/A
Multiple Restart Next Step Hill Climber	3	Bin Pack 2, Griewangk, Order-4 Deceptive

There was no clear best within the PBIL2 category. However, on the problems on which PBIL2 performed the best, a setting of 0.075 for the negative learning rate parameter was able to do better than PBIL, PBIL-N, PBIL-L, SGA and the Hill-Climber. Nonetheless, the optimal setting of the negative learning rate is very problem dependent. For example, on several problems, Jobshop 10x10, Jobshop 20x5, and the Checkerboard problem, the performance of PBIL2 was improved with alternate settings of the negative learning rate.

The results achieved by PBIL are more accurate and are attained faster than a conventional genetic algorithm. For example, the average final result obtained by the SGA for the 20x5 jobshop problem was attained in generation 1702 (after this generation, no improvement was made), PBIL2 (0.075) attained the same quality solution at generation 191. This type of speedup is found in many of the problems attempted. The PBIL-2 (0.075) algorithm was able to find equivalent solutions to the SGA, measured over the entire run, in fewer generations. Table III shows in which generation the SGA was able to first achieve its highest evaluation, and in which generation PBIL-2 (0.075) was first able to surpass it. The table also shows in which generation the S-PBIL algorithm was able to achieve its highest evaluation, and in which generation the PBIL-2 (0.075) algorithm was able to surpass it. On the De Jong functions (F2 and F3) and the order-4 deceptive problems, all algorithms were able to reach approximately the same solutions.

Table III: Relative Performance of PBIL2 (0.075) compared with SGA and S-PBIL

PROBLEM	Generation in which Highest SGA Evaluation was First Found. (Generation)	Generation in which PBIL2 (0.075) surpassed highest SGA Evaluation. (Generation)	Generation in which Highest S-PBIL Evaluation was First Found. (Generations)	Generation in which PBIL2 (0.075) surpassed highest S-PBIL Evaluation. (Generations)
Jobshop 10x10	1894	215	1039	353
Jobshop 20x5	1702	191	1271	1015
TSP 30	1928	219	1916	1164
TSP 50	1954	148	1961	444
Bin Pack 1	1982	1039	1986	784
Bin Pack 2	1991	1870	1988	1954
Bin Pack 3	1949	932	1930	968
Griewangk	1962	160	1977	200
Checkerboard	1985	156	1994	1173

6. Conclusions

The simple population-based incremental learner performs better than a standard genetic algorithm in the problems empirically compared in this paper. The results achieved by PBIL are more accurate and are attained faster than a standard genetic algorithm, both in terms of the number of evaluations performed and the clock speed. The gains in clock speed were achieved because of the simplicity of the algorithm. The algorithm does not require all the mechanisms of a GA; rather the few steps in the algorithm are small and simple. For example, the implementation explored for this study - with all of the surveyed extensions - totaled less than 100 lines of C code.

One explanation for the success of the PBIL algorithm can be attributed to its ability to focus search effort in one region of the space very quickly. In comparing the ability of the GA and of PBIL to focus search, the

GA was able to maintain diversity longer than PBIL (for some settings of the learning rate). Nonetheless, the GAs ability to postpone commitment to one region of the function space does not allow it to extensively explore any region. When one region is explored extensively, the GA's population loses diversity, and thereby also loses its ability to explore other regions. The time to make the commitment to one region is longer in GAs than in PBIL. However, the empirical results show that PBIL does enough exploration before the commitment is made to outperform a conventional GA. Therefore, this postponement of commitment in the GA, at least on the problems attempted here, is not necessary. This line of explanation is under study.

Another benefit over traditional genetic algorithms is that PBIL is easily parallelizable. Parallelization of this algorithm on both coarse and fine grain parallel machines is simply implemented. Although GAs have also been parallelized on both types of machines, many fine grain parallelization methods incur a large amount of inter-process communication, which can severely affect the time performance of the algorithm. For example, unless the GA incorporates a subpopulation structure, a single synchronizing processor must handle the pairing of the solution strings, and control the distribution of the solution strings from their original processors to their new processors. Although a subpopulation model will ease the communication overhead, both algorithms, PBIL and the subpopulation GA, can benefit from the reduced communication costs in this model. In the parallelization of a single population PBIL, the inter-processor communication is minimal, as only the highest and lowest evaluation vector generated are returned to the synchronizing processor. The updated probability vector is then sent to all of the processors. All of the surveyed modifications to the original PBIL algorithm also incur very little overhead. In terms of the number of function evaluations, the modifications are free. Computationally, the additions do not add noticeable time penalties.

It is clear from the results that using negative examples helps to achieve better results. However, there is no clear winner regarding the setting of the negative learning rate. The best setting of the negative learning rate varies per problem. A much more focused study should be done on each of the parameters. Particular attention was given to the negative learning rate in this paper as this is apparently the parameter which has the least amount of literature available.

The empirical results presented here indicate that where a standard genetic algorithm does well, this algorithm should also do as well or better. However, on problems on which genetic algorithms fail, this technique may also perform badly. Although the mechanisms of PBIL are different than those usually employed in GA literature, standard PBIL may still be susceptible to some of the suitability issues associated with genetic optimization. For example, the bin packing problems have good approximation algorithms [Garey and Johnson, 1979], and in fact good solutions can be found in many cases with hill-climbing or simple depth-first search in reasonable amounts of time. This leads to questions about whether the genetic optimization schemes are appropriate for this class of problems. This paper does not address this issue, as it has been a topic of study for many researchers in this area. The scope of this paper is limited to improving the optimization capabilities of GAs, not selecting the problems to which GAs should be applied.

Although the algorithm presented here is a significant departure from standard genetic algorithms and competitive learning, both of the fields have contributed enormously to the final algorithm. The competitive learning facet of the algorithm was substantially modified from the majority of CL algorithms and applications found in current literature. First, the aim of the algorithm was to find a prototype vector for the class of high evaluation vectors, which is a supervised task. Second, the training of the CL-net also directed the progression of which points would be seen by the CL- net; usually the training of the CL-net does not effect the points which are to be used in the training. Finally, the class of interest, the high evaluation vectors, was not pre-defined. Rather, the class was defined relative to the currently generated population of points. The highest evaluation vector in the current population was defined to be in the class of interest.

Basic genetic algorithm features such as a population and the crossover operator were implemented in quite a different manner than the simple GA envisioned by Holland and De Jong [Holland, 1975] [De Jong, 1975]. A very basic intuition of the progress made by a GA, and its counterpart in PBIL, was briefly given in this paper. One of the difficulties in standard genetic algorithms which this algorithm eliminates is the issue of scaling. In the beginning of this paper, it was mentioned that a GA needs the function to be appropriately scaled to perform accurate optimization. This algorithm does not rely on any specific form of scaling, it only requires that the scaling is increasing for increasing fitness. The scaling is inherently relative, and only requires the best and worst evaluations in a population. Rank-based GAs have also been described in literature; this approach may yield improved performance of the GA on the test problems.

The basic algorithm, as it is presented in this paper, is very simple. In the same way that GAs have enjoyed a great deal of research increasing their effectiveness for a variety of interest-specific goals, similar additions can be made for PBIL. In this paper, only a standard GA was considered. A GA with different mechanisms, such as non-stationary mutation rates, local optimization heuristics, parallel subpopulations, specialized crossover, or larger alphabets, may perform better. However, all of these mechanisms, with the exception of specialized crossover operators, can be used with PBIL with no modifications; many have been implemented with very promising results. The claim is not that the basic PBIL will be able to outperform all GAs; more work needs to be conducted to determine where each method will prove to be more beneficial. The hope is that the simple framework of PBIL will be used as an underlying model from which extensions and insights can be formed.

7. Future Directions

7.1. Non-Binary Encodings

Although the theoretical studies of GAs suggest using a low cardinality encoding for the solution strings, practice has often revealed good solutions through more natural encodings. In a higher cardinality set, the probability of each position can be represented by a set of probabilities for generating each possible member at each location in the solution vector.

Many of the problems attacked in genetic algorithm literature are continuous valued optimization functions which have been discretized to an arbitrary value. A possible method of generating continuous values is to specify a gaussian curve over the desired range. The average and the variance for each variable can be “evolved” during the search. Methods of encoding solutions which are similar to this are often used in evolutionary strategies and evolutionary programming [Baeck & Schwefel, 1993].

7.2. Drawing from Genetic Algorithm Literature

7.2.1. Multiple Populations

In genetic algorithms research, there is a growing trend towards using parallel genetic algorithms. As mentioned before, these algorithms explicitly maintain parallelism. Similar techniques are possible in PBIL. Multiple prototype vectors can be formed in parallel, each generating its own set of potential solutions. In order to synthesize the search in each of the populations, positions in the probability vectors can be swapped [Juels, 1993][Gordon, 1993]. Preliminary experiments have been tried, and have revealed improved results. Although this superficially appears to be more closely related to the LVQ algorithm as multiple prototype vectors are developed, this is not the case. Each prototype vector is not trying to distinguish unique classes, rather all of the vectors are used to define only the single class of interest.

7.2.2. Time Varying Mutation Rates

One of the current areas of research in the GA community is the use of adaptive, or time-varying, mutation rates. These provide the ability to control the amount of mutation based upon the convergence of the population. The need for mutation is largest when the population has converged. Convergence in the PBIL can be measured by the similarity of the vectors generated, or directly by the values of the prototype vector. Mutation rates can be increased in the latter portions of search when the vectors generated may be too homogenous. As in GAs, this may aid in escaping local optima by maintaining a heterogenous population.

7.2.3. Intelligent Mutation

Currently, mutation is implemented by selecting a position to mutate, then moving the value in the probability vector a specified distance in a random direction (either towards 0.0 or 1.0). As the primary goal of mutation is to preserve diversity, perhaps a more intelligent method of using mutation is to move the selected position towards a less committed state: 0.5. This ensures that the mutation increases the amount of diversity in the next population.

7.2.4. Elitist Selection

In standard genetic algorithms, there is no guarantee that once a good solution vector is found that it will remain in the population of the subsequent generation. For example, it may not be selected for recombination, as selection is probabilistic, or it may be altered by mutation and/or crossover. A method used to resolve this is to carry the best solution vector from one generation to the next unaltered. Similarly, in the PBIL algorithm, it is possible that although a good solution vector is found in one generation, it may not be found again in the next, as the generation of solution vectors is probabilistic. In a manner similar to elitist selection, the best solution vector from the previous generation can be placed into the current generation. The solution vector will only be chosen again for updating the probability vector if no better solution is found.

7.3. Drawing from Artificial Neural Network (ANN) Literature

7.3.1. Competition of Probability Vectors

This paper has presented a method for replacing the population of a GA by a single probability vector. The update rule for the probability vector is derived from competitive learning. However, it may be possible to derive more than the update rule from the principles of competitive learning. Multiple probability vectors can be used to model a population. The next generation can be created by sampling each probability vector for a subset of the next population. Multiple good solution vectors can be used to update the probability vectors. Each good solution vector can update the probability vector to which it is most similar, as is done in standard competitive learning. This method is also currently under study by [Baluja & Boyan, 1994].

7.3.2. Adapting the Learning Rate

A method that has been proposed in the ANN literature to ensure that the prototype vectors in a competitive learning network stabilize is to lower the learning rate slowly to 0.0. However, non-convergence has not been a problem in this algorithm. Rather, a large problem is avoiding premature convergence. As the learning rate controls the amount of exploration vs. the amount of exploitation the algorithm is performing, adjusting the learning rate during the course of a training session may provide increased ability to explore diverse regions of the function space.

7.3.3. Explicitly Avoiding Premature Convergence of Probabilities

One of the methods suggested by Fahlman to aid in the training of artificial neural networks is to ensure that the outputs of the neurons in an ANN are not pinned to extreme values [Fahlman, 1989]. This is not an immediate problem in PBIL, as the update rule ensures that when the values of a probability begin to reach an extreme, a move away from the extreme is larger than a move towards the extreme. However, only allowing the probabilities to come within a pre-specified distance of the extremes, as Fahlman [Fahlman, 1989] has done, may also aid in avoiding local optima.

7.3.4. Using Noise to Avoid Premature Convergence of Probabilities

A method used in traditional competitive learning to ensure that all output units are eventually activated is to add noise to the pattern vectors [Hertz, Krogh &Palmer, 1993]. In PBIL, once the highest evaluation vector is determined in the current generation, a small amount of noise can be added to the vector before it is used to update the prototype vector. In the few problems tested with this technique, the results were improved over optimization without noise. Nonetheless, the effectiveness of this measure must be more carefully studied before firm conclusions can be made.

7.4. Choosing the Appropriate Problems

One of the future directions which should immediately be considered is not only relevant to PBIL, but also to the field of genetic algorithms. The problems chosen here were static function optimization problems. They were chosen because either the exact same problems or problems very similar to these appear in a large amount of GA research. Although much of the GA research has been limited to the type of static function optimization presented here, simpler heuristics, ranging from PBIL to restart-hill- climbing methods, have been shown empirically to be as effective as genetic algorithms. For example, in this paper alone, on three of the twelve problems attempted (Bin Packing 2, Griewangk, and Order-4b) the hillclimbing procedure performs the best. Further, if the hillclimbing algorithm were a bit more sophisticated and included moves to regions of equal evaluation (this would also necessitate the need for cycle detection for restart) the hill-climbing performance might be improved on some of the problems attempted here. This reflects the tremendous need to more carefully study and define the domains in which GAs, and evolutionary algorithms in general, yield benefit, in light of their very high computational burden.

8. Acknowledgments

The ideas presented in this paper would not have been possible without the months of discussions and work with Ari Juels on the basic PBIL and EGA. Ari Juels' diligent pursuit of the more theoretical ramifications of the EGA on GA based search has been a large motivation for the study of PBIL in terms of competitive learning. Special thanks are also owed to Geoff Gordon who has helped throughout the development of this paper, from the initial concepts to reviewing drafts. The paper has also greatly benefited from the reviews of David Fogel, Worthy Martin, Andrew Moore, Justin Boyan, and Dean Pomerleau.

The author is supported by a National Science Foundation Graduate Fellowship. This research was supported by the Department of Navy, Office of Naval Research under Grant No. N00014-93-1-0806. The views and conclusions contained in this document are those of the author and should not be interpreted as representing official policies, either expressed or implied, of the National Science Foundation or the U.S. Government.

9. References

Ackley, D.H. (1985) "A Connectionist Algorithm for Genetic Search". In Greftenstette (ed.) *Proceedings of an International Conference on Genetic Algorithms and their Applications*. Greftenstette.

Ackley, D.H. (1987) *A Connectionist Machine for Genetic Hillclimbing*. Kluwer Academic Publishers, Boston, MA.

Baeck, T. (1993), "Optimal Mutation Rates in Genetic Search". In Forrest (ed). *Proceedings of the Fifth International Conference on Genetic Algorithms*. 2-8. Morgan Kaufmann Publishers.

Baeck, T. & Schwefel, H.P. (1993), "An Overview of Evolutionary Algorithms for Parameter Optimization", *Evolutionary Computation*, Vol 1:1, p.1.

Baluja, S. (1993) "Structure and Performance of Fine-Grain Parallelism in Genetic Search". In Forrest (ed). *Proceedings of the Fifth International Conference on Genetic Algorithms*. 155-162. Morgan Kaufmann Publishers. San Mateo, CA.

Baluja, S. & Boyan, J. (1994) "Parallel Search - Genetic Algorithms and Supervised Competitive Learning". Paper in Progress.

Booker, L.B., Goldberg, D.E., Holland, J.H. (1990), "Classifier Systems and Genetic Algorithms". In Shavlik, J.W. & Dietterich, T. (ed.) *Readings in Machine Learning*. 404-429. Morgan Kaufmann Publishers, San Mateo, CA.

Boyan, Justin (1993). Personal Communication, 1993.

Caruana, R. & Schaffer, J. (1988), "Representation and Hidden Bias: Gray vs. Binary Coding for Genetic Algorithms". In Laird, J. (ed.) *Proceedings of the Fifth International Conference on Machine Learning*. 153-161. Morgan Kaufmann Publishers. San Mateo, CA.

Cohoon, J.P, Hedge, S.U., Martin, W.N., Richards, D. (1988), "Distributed Genetic Algorithms for the Floor Plan Design Problem". Technical Report TR-88-12. School of Engineering and Applied Science, Computer Science Department, University of Virginia.

Davidor, Y., Yamada, T. & Nakano, R. (1993) "The ECOlogical Framework II: Improving GA Performance At Virtually Zero Cost". In Forrest (ed). *Proceedings of the Fifth International Conference on Genetic Algorithms*. 171 - 176. Morgan Kaufmann Publishers. San Mateo, CA.

Davis, L. (1991) "Bit Climbing, Representational Bias, and Test Suite Design". In Belew and Booker (eds.) *Proceedings of the Fourth International Conference on Genetic Algorithms*. 18-23. 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.) *Foundations of Genetic Algorithms-2*. 5-17. Morgan Kaufmann Publishers. San Mateo, CA.

De Jong, K., Spears, W.M., Gordon, D., (1993) "Using GAs for Concept Learning", *Machine Learning* V.13 2-3: 161-188.

Duda, R.O. and Hart, P.E. (1973). Pattern Classification and Scene Analysis. Wiley.

Eshelman, L.J. (1991) "The CHC Adaptive Search Algorithm: How to Have Safe Search When Engaging in Nontraditional Genetic Recombination". In Rawlins, G.J. (ed.) *Foundations of Genetic Algorithms*. 265-283. Morgan Kaufmann, San Mateo, CA.

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

Fahlman, S.E. (1989) "Fast-Learning Variations on Back-Propagation: An Empirical Study". In Touretzky, Hinton, Sejnowski (ed.) *Proceedings of the 1988 Connectionist Models Summer School.* 38-51. Morgan Kaufmann, 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 (ed). *Proceedings of the Fifth International Conference on Genetic Algorithms*. 375-382. Morgan Kaufmann Publishers. San Mateo, CA.

Fogel, D. B. (1994) "An Introduction to Simulated Evolutionary Optimization". *IEEE Neural Networks*. Vol. 5, NO.1. January, 1994.

Fogel, D. B. & Stayton, L.C. (1993) "On the Effectiveness of Crossover in Simulated Evolutionary Optimization", in press. BioSystems Journal.

Garey, M. & Johnson, D. (1979) *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman and Co.

Goldberg, D.E. & Richardson, J. (1987) "Genetic Algorithms with Sharing for Multimodal Function Optimization". In Greftenstette (ed.) *Proceedings of the Second International Conference on Genetic Algorithms*. Morgan Kaufmann, San Mateo, CA.

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

Goldberg, D.E., Kalyanmoy, D. & Clark, J.H., (1992) "Genetic Algorithms, Noise and the Sizing of Populations". *Complex Systems* 6: 333-362.

Gordon, Geoffery (1993). Personal Communication, 1993, 1994.

Gordon, V.S., Whitley, D. (1993) "Serial and Parallel Genetic Algorithms as Function Optimizers". In Forrest (ed). *Proceedings of the Fifth International Conference on Genetic Algorithms.* 177-183. Morgan Kaufmann Publishers. San Mateo, CA.

Gray, R.M. (1984) Vector Quantization. *IEEE ASSP Magazine*, April 19844-29.

Greftenstette, J. (1992) "Deception Considered Harmful". In Whitley (ed.) *Foundations of Genetic Algorithms-2*. 75-91. Morgan Kaufmann Publishers. San Mateo, CA.

Gruau, F. (1993) "Genetic Synthesis of Modular Neural Networks". In Forrest (ed). *Proceedings of the Fifth International Conference on Genetic Algorithms*. 318-325. Morgan Kaufmann Publishers.

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

Holland J. (1992) *Adaptation in Natural and Artificial Systems.* MIT Press, Cambridge, MA. 1992.

Hopfield, J. & Tank, D.W. (1985) "'Neural' Computation of Decisions in Optimization Problems". *Biological Cybernetics* 52, 141-152.

Janikow, C.Z. (1993) "A Knowledge Intensive GA for Supervised Learning". *Machine Learning* V.13 2-3: 187-228.

Juels, Ari. (1993, 1994) Personal Communication.

Kaelbling, L. P. (1990) *Learning in Embedded Systems*. Ph.D. Thesis, Stanford University.

Kirkpatrick, S., Gelatt, C.D. & Vechi, M.P. (1983). "Optimization by Simulated Annealing" *Science* 220. 671-680

Kohonen, T. (1990) "Improved Versions of Learning Vector Quantization". Proceedings of the International Joint Conference on Neural Networks vol. 1. 545-550. IEEE.

Kohonen, T. (1989) *Self-Organizing and Associate Memory* (3rd ed.). Springer Verlag. Berlin.

Kohonen, T., Barna, G., Chrisley, R. (1988). "Statistical Pattern Recognition with Neural Networks: Benchmarking Studies". In *IEEE International Conference on Neural Networks*, vol. I, 61-68. New York: IEEE.

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

Le Cun, Y. et al (1990) "Handwritten Digit Recognition with a Back- Propagation Network" D.S. Touretzky (ed). *Advances in Neural Information Processing Systems II*. Morgan Kaufmann. San Mateo, CA.

Liepins, G.E. & Vose, M.D. (1990) "Representational Issues in Genetic Optimization". *Journal of Experimental and Theoretical Artificial Intelligence* 2: 101-115.

Moore, Andrew (1994), Personal Communication.

Muhlenbein, H. (1989), "Parallel Genetic Algorithms, Population Genetics and Combinatorial Optimization." In Schaffer (ed.) *Proceedings of the Third International Conference on Genetic Algorithms*. 416-421. Morgan Kaufmann, San Mateo, CA.

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

Polani, D. & Uthmann, T. (1993) "Training Kohonen Feature Maps in different Topologies: an Analysis using Genetic Algorithms". In Forrest (ed). *Proceedings of the Fifth International Conference on Genetic Algorithms.* 326-333. Morgan Kaufmann Publishers. San Mateo, CA.

Pomerleau, D.A. (1989). "ALVINN: An Autonomous Land Vehicle in a Neural Network." In D.S. Touretzky (ed). *Advances in Neural Information Processing Systems I*. Morgan Kaufmann. San Mateo, CA.

Schleuter, M.G. (1990), "Explicit Parallelism of Genetic Algorithms through Population Structures". In Schwefel and Manner (eds.) *Parallel Problem Solving from Nature*. 150-159. Springer-Verlag. Berlin.

Spears, W.M. (1992) "Crossover or Mutation?". In Whitley (ed.) *Foundations of Genetic Algorithms-2.* 221-237. Morgan Kaufmann Publishers. San Mateo, CA.

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

Waibel, A., Hanzawa, T., Hinton, G., Shikano, K. & Lang, K. (1989). "Phoneme Recognition Using Time-Delay Networks". *IEEE Transactions on Acoustics, Speech and Signal Processing* 37, 328-339.

Whitley, D. & Hanson, T. (1989) "Optimizing Neural Networks using Faster, more Accurate Genetic Search". In D. Schaffer (ed.) *Proceedings of the Third International Conference on Genetic Algorithms.* 391-396. Morgan Kaufmann Publishers. San Mateo, CA.

Whitley, D. & Schaffer, D. (1992) Eds. *Proceedings of the International Workshop on Combinations of Genetic Algorithms and Neural Networks*. IEEE Computer Society.

Whitley, D., Starkweather, T. & Fuquay, D., (1989) "Scheduling Problems and Tavelling Salesmen: The Genetic Edge Recombination Operator". In J.D. Schaffer (ed.) *Proceedings of the Third International Conference on Genetic Algorithms.* Morgan Kaufmann, San Mateo, CA.

Whitley, D. & Starkweather, T. (1990) "GENITOR II: A distributed Genetic Algorithm". *Journal of Experimental and Theoretical Artificial Intelligence* 2: 189-214.

10. Appendix A

The typical crossover operators which are used in standard genetic algorithms are described here. These operators are most commonly implemented with GAs which have fixed length potential solution representations. Other crossover operators have been developed for problems in which the solution representations may not be fixed in length, such as in the task of genetic programming [Koza, 1992]. Other operators have also been developed to take advantage of specific characteristics of the problem being addressed. The three crossover operators described here are general purpose recombination operators which have been used and studied widely in genetic algorithm literature.

One Point Crossover:

Given two parent chromosomes A & B, select a randomly chosen crossover point, swap contents of the chromosomes beyond the chosen point:

```
Parent A      00000000000 00000
Parent B      11111111111 11111

Child A       00000000000 11111
Child B       11111111111 00000
```

Two Point Crossover:

Given two parent chromosomes A & B, select two randomly chosen crossover points, swap contents of the chromosomes between chosen points:

```
Parent A      000 0000000000 000
Parent B      111 1111111111 111

Child A       000 1111111111 000
Child B       111 0000000000 111
```

Uniform Crossover:

Given two parent chromosomes A & B, select from either parent randomly:

```
Parent A      0000000000000000
Parent B      1111111111111111

Child A       0101011010111000
Child B       1010100101000111
```

11. Appendix B: The Hillclimbing Algorithm

In order to judge the performance of the GA and PBIL algorithms, the problems are also attempted using hill-climbing methods. The hill-climbing algorithm is tested on the same problem representations as were used in the GA and PBIL algorithms. The algorithm is shown in Figure 19.

```
L ⟵ clear list

V ⟵ randomly generate solution vector
Best ⟵ evaluate (v)

loop # ITERATIONS

        pos ⟵ random_integer (1..LENGTH) ∉ L
        V_pos ⟵ Flip_Bit (V_pos)
        V_eval ⟵ evaluate (V)
        if (V_eval > Best)
                Best ⟵ V_eval
                L ⟵ clear list
        else
                V_pos ⟵ Flip_Bit (V_pos)
                L ⟵ add pos to L
```

USER DEFINED CONSTANTS:
LENGTH: the length of the solution encoding (in bits)
ITERATIONS: how many evaluations are to take place throughout the run of the algorithm

VARIABLES:
V: the current solution vector
V_eval: the current evaluation.
Best: the best evaluation ever found.
L: the list of moves attempted which resulted in evaluations ≤ best.

Figure 19: The hillclimbing algorithm used. It is shown for solution vectors represented in binary. In the full algorithm, the best vector along with its evaluation would be saved.

12. Appendix C: A Simple Genetic Algorithm

This appendix gives a small outline of a basic genetic algorithm. Of course, many variations are possible, and have been explored in literature. The GA shown in Figure 20 is generational, and every solution vector in the previous generation is replaced by new solution vectors. The GA also uses a modest form of elitist selection, in which the best solution vector from the previous generation replaces the worst solution vector in the current generation.

```
i ⟵ loop # POPULATION_SIZE
        PopulationA_i ⟵ randomly generate a solution vector
        EvaluationA_i ⟵ evaluate (PopulationA_i)
        best ⟵ find highest evaluation in EvaluationA

loop # GENERATIONS
        loop # POPULATION_SIZE/2
                one ⟵ select vector probabilistically, based upon evaluation
                two ⟵ select vector probabilistically, based upon evaluation
                child1, child2 ⟵ generate two children,
                                based on crossover of (PopulationA_one, Population A_two)
                child1 ⟵ perform mutation (child1, MUTATION_RATE)
                child2 ⟵ perform mutation (child2, MUTATION_RATE)
                PopulationB ⟵ add child1 to PopulationB
                PopulationB ⟵ add child2 to PopulationB

        i ⟵ loop # POPULATION_SIZE
                EvaluationB_i ⟵ evaluate (PopulationB_i)

        worst ⟵ find vector corresponding to the worst evaluation in Population B

        PopulationB_worst ⟵ PopulationA_best

        PopulationA ⟵ PopulationB
        EvaluationA ⟵ EvaluationB
        best ⟵ find highest in PopulationA
```

USER DEFINED CONSTANTS

GENERATIONS: the number of generations the algorithm is allowed to continue (2000)
POPULATION_SIZE: the number of samples generated per generation (100)
MUTATION_RATE: the probability of flipping each bit (0.001)

IMPLICIT DECISIONS (CONSTANTS NOT SHOWN HERE):

CROSSOVER RATE: How many vectors to replace in subsequent generations (Every solution vector is replaced)
CROSSOVER TYPE: 2 pt, 1pt, Uniform, Specialized, etc. (2 pt)
ELITIST-SAVES: how many solution vectors are saved from the previous generation (1)
LENGTH: the length of the solution encoding (Problem specific)

VARIABLES:

PopulationA, PopulationB: arrays of solution vectors. Arrays of size POPULATION_SIZE
EvaluationA, EvaluationB: arrays evaluations of the solution vectors in Population A & B, respectively.
child1, child2: two solution vectors produced by the crossover operations.
best, worst: the best and worst in a population, they are based upon the vector's evaluation.

Figure 20: The basic GA algorithm used in this study. The number in parentheses are the settings used for the empirical studies in this paper.