# Stochastic Hillclimbing as a Baseline Method for Evaluating Genetic Algorithms

Ari Juels* Martin Wattenberg†

September 28, 1994

**Abstract**

We investigate the effectiveness of stochastic hillclimbing as a baseline for evaluating the performance of genetic algorithms (GAs) as combinatorial function optimizers. In particular, we address four problems to which GAs have been applied in the literature: the maximum cut problem, Koza's 11-multiplexer problem, MDAP (the Multiprocessor Document Allocation Problem), and the jobshop problem. We demonstrate that simple stochastic hillclimbing methods are able to achieve results comparable or superior to those obtained by the GAs designed to address these four problems. We further illustrate, in the case of the jobshop problem, how insights obtained in the formulation of a stochastic hillclimbing algorithm can lead to improvements in the encoding used by a GA.

*Department of Computer Science, University of California at Berkeley. Supported by a NASA Graduate Fellowship. This paper was written while the author was a visiting researcher at the Ecole Normale Supérieure–rue d'Ulm, Groupe de BioInformatique, France. E-mail: **juels@cs.berkeley.edu**

†Department of Mathematics, University of California at Berkeley. Supported by an NDSEG Graduate Fellowship. E-mail: **wattenbe@math.berkeley.edu**

# 1 Introduction

*Genetic algorithms* (GAs) are a class of randomized optimization heuristics based loosely on the biological paradigm of natural selection. Among other proposed applications, they have been widely advocated in recent years as a general method for obtaining approximate solutions to hard combinatorial optimization problems using a minimum of information about the mathematical structure of these problems. By making use of a general "evolutionary" strategy, GAs aim to maximize an objective or *fitness* function $f : S \rightarrow \mathbf{R}$ over a combinatorial space $S$, i.e., to find some state $s \in S$ for which $f(s)$ is as large as possible. (The case in which $f$ is to be minimized is clearly symmetrical.) For a detailed description of the algorithm see, for example, [14], which constitutes a standard text on the subject.

GAs belong to a larger class of methods known as *black-box algorithms* – i.e., algorithms which attempt to optimize a function using a strategy essentially independent of the problem at hand. As a black-box algorithm, the GA requires very little knowledge about the combinatorial structure of the problem to be solved, so it is naturally applied to problems whose structure is poorly understood. In particular, since the optima of these problems are generally unknown, it is difficult to choose criteria according to which to assess the GA's performance. Researchers often resort to comparisons of the GA with extremely naïve methods such as random search, or else simply leave the GA to stand on its own. Informed comparisons with other black-box methods, such as simulated annealing and hillclimbing, are surprisingly rare.

In this paper, we investigate the effectiveness of the GA in comparison with that of stochastic hillclimbing (SH), a probabilistic variant of hillclimbing. As the term "hillclimbing" suggests, if we view an optimization problem as a "landscape" in which each point corresponds to a solution $s$ and the "height" of the point corresponds to the fitness of the solution, $f(s)$, then hillclimbing aims to ascend to a peak by repeatedly moving to an adjacent state with a higher fitness. Hillclimbing represents a very naïve black-box method, particularly in comparison with genetic algorithms or simulated annealing. We would hope at the very least therefore that, in justification of its greater complexity, the GA would be able to outperform SH on hard combinatorial optimization problems. This is not always the case, however. As we attempt to demonstrate in this paper, SH proves a more effective optimization method on certain problems which have received a great deal of attention in the literature as demonstrating areas in which GAs perform strongly. In consequence, we advocate a more widespread use of SH as a basic litmus test for the performance of the GA.

A number of researchers in the GA community have already addressed the issue of how various versions of hillclimbing on the space of bitstrings, $\{0,1\}^n$, compare with GAs. An early effort in this direction is to be found in the thesis of Ackley [1], which considers several varieties of SH on bitstrings in comparison with a method called "stochastic iterated genetic hillclimbing" (SIGH). Ackley performs his experiments on a suite of test functions of his own devising and a collection of instances of the minimum-cut graph partition problem. Davis [8] makes the surprising discovery that hillclimbing on bitstrings outperforms standard GAs on the widely referenced De Jong suite of functions [19], and proposes a modification of these functions to make them more difficult for the simpler algorithm. Wilson [31] points out the existence of some very elementary functions on which GAs can outperform a particular type of hillclimbing called steepest-ascent. Mitchell and Holland [25] present an extensive analysis of the relative performances of bitstring-based hillclimbing and GAs on a synthetic function known as the "Royal Road".

Our investigations in this paper differ in two important respects from these previous ones. First, we address more sophisticated problems than the majority of these studies, which develop test functions exclusively for the purpose of exploring certain state-space characteristics. We examine in this paper three classical NP-complete problems and a genetic programming problem. Second, we consider hillclimbing algorithms based on operators in some way "natural" to the combinatorial structures of the problems to which we are seeking solutions, very much as GA designers attempt to do. In several cases, our SH algorithms employ exactly the same encoding as proposed GAs, and adopt operators which correspond exactly to these GAs' mutation operators. Consequently, in three of the four problems in this paper, the hillclimbing algorithms we consider operate on structures other than bitstrings.

The remainder of the paper is organized as follows. We describe in Section 2 the form of the SH algorithm used in the experiments in this paper and provide details regarding this algorithm's implementation. In Section 3, we consider four problems for which GA based approaches have received important attention in the literature. These are the maximum cut problem, Koza's 11-Multiplexer problem, MDAP (the Multiprocessor Document Allocation Problem), and the jobshop problem. We design SH algorithms for these problems, and present the results of experiments demonstrating the superiority of these SH algorithms to the GAs proposed in the literature. In addition, for the jobshop problem, we show how were able to improve a GA approach to the problem by making use of the SH algorithm as a basis for its design. We summarize in Section 4, arguing in favor of SH methods as a simple litmus test to determine whether or not the relatively complex apparatus of the GA is justified for a given problem.

## 2 Stochastic Hillclimbing

The SH algorithm employed in this paper searches a discrete space $S$ with the aim of finding a state whose fitness is as high (or as low) as possible. The algorithm does this by making successive improvements to some current state $\sigma \in S$. As is the case with genetic algorithms, the form of the states in $S$ depends upon how the designer of the SH algorithm chooses to encode the solutions to the problems to be solved: as bitstrings, permutations, or in some other form. The local improvements effected by the SH algorithm are determined by the *neighborhood structure* and the fitness function $f$ imposed on $S$ in the design of the algorithm. We can consider the neighborhood structure as an undirected graph $G$ on vertex set $S$. The algorithm attempts to improve its current state $\sigma$ by making a transition to one of the neighbors of $\sigma$ in $G$. In particular, the algorithm chooses a state $\tau$ according to some suitable probability distribution on the neighbors of $\sigma$. If the fitness of $\tau$ is as least as good as that of $\sigma$ then $\tau$ becomes the new current state, otherwise $\sigma$ is retained. This process is then repeated. The algorithm essentially performs a random walk in which moves leading to a reduced fitness in the current state are rejected. More formally, in the form of a piece of pseudocode performing SH for $M$ iterations:

```
σ ∈_R S;
FOR J = 1 TO M
        τ ∈_R N(σ);
        IF f(τ) ≥ f(σ) THEN
                σ = τ;
```

Here $\in_R$ indicates random selection over some suitable distribution (which does not generally depend upon fitness), while $N(\sigma)$ indicates the set of neighbors of $\sigma$.[1]

Of course, the possible variations on this algorithm are numerous. A difference between the version presented here and those frequently used in practice is that when neighboring state $\tau$ is selected, it is made the new current state when the fitness of $\tau$ is greater than *or equal* to that of $\sigma$, rather than merely greater. This apparently trifling but in fact significant type of variation often goes unconsidered in implementations of hillclimbing, even in detailed and comprehensive studies of function optimization heuristics such as [17] and [18]. In the formulation in this paper, the hillclimbing algorithm is able to explore "level surfaces," i.e., regions in the search space in which the fitnesses of neighboring states are equal. In practice, this appears to lead to more rapid discovery of high fitness solutions. If we required instead that $f(\tau) > f(\sigma)$, the algorithm could never improve its current state if $\sigma$ were located in the middle of a level surface.

Another possible generalization involves a consideration of the tradeoff between the length of individual hillclimbing runs and the number of runs performed in a single experiment. In the case of jobshop, we execute the hillclimbing algorithm multiple times for short durations rather than in one single run, and find that this improves the performance of the algorithm considerably. We do not explore the issue in any depth in this paper, but it is potentially an interesting avenue for future investigation.

[1]This algorithm can be viewed as simulated annealing with the temperature held constant at 0. The method of annealing at a fixed temperature is sometimes referred to as the *Metropolis process*. At temperature 0, all transitions leading to states with a "higher energy" (inferior fitness) are rejected, and all others accepted with probability 1. At higher temperatures, simulated annealing moves to inferior states $I$ with a probability that depends on the difference $f(\sigma) - f(I)$ and the current temperature.

# 3 Maximum Cut, MDAP, GP, and Jobshop

## 3.1 The Experiments

In this section, we compare the performance of SH algorithms with that of GAs proposed for four problems: the maximum cut problem, the jobshop problem, MDAP (the Multiple Document Allocation Problem), and Koza's 11-multiplexer problem. We choose these four problems in particular because they have received considerable attention as examples on which GAs perform well. Most of the instances of the problems on which we conduct our comparisons are drawn directly from the articles in which the GA approaches to these problems are reported.

We gauge the performance of the GA and SH algorithms according to the fitness of the best solution achieved after a fixed number of function evaluations, rather than the running time of the algorithms. This is because evaluation of the fitness function generally constitutes the most substantial portion of the execution time of the optimization algorithm, and accords with standard practice in the GA community.

## 3.2 Maximum Cut

In order to demonstrate the effectiveness of the GA as a general combinatorial optimization method, Khuri, Bäck, and Heitkötter [20] apply the algorithm to several NP-complete problems. We compare their results on one of these, the maximum cut problem, with results obtained by an SH approach.

In its usual formulation, the maximum cut problem takes as inputs an undirected graph $G = (V, E)$ and a set of positive, integer weights $W$ on $E$. The objective of the problem is to partition $V$ into two sets of vertices, $V_0$ and $V_1$, so as to maximize the total weight of the edges whose endpoints lie in different sets – i.e., edges in $V_0 \times V_1$. Maximum cut is NP-complete [13].

**The GA** Khuri et al. encode a partition for this problem in the form of a bitstring $B = (x_1, x_2, \ldots, x_n)$, where $n$ is the number of vertices in $G$. This bitstring $B$ specifies a partition in which each vertex $i$ is assigned to set $V_{x_i}$.

The GA applied to this encoding employs one-point crossover with a crossover rate of 0.6 and a mutation rate of $\frac{1}{n}$, where $n$ is the length of the bitstrings on which the algorithm operates. Selection follows a method known as linear-dynamical scaling (see [14] pp. 123-4). Except in the case of the mutation rate, for which the authors adhere to a different standard, all parameters in this GA are identical to those in Greffenstette's GENESIS software package [16].

**The SH Algorithm** In this problem, changing a single bit corresponds to the combinatorially natural operation of moving a single vertex from one set in the partition to the other. We therefore employ the same encoding as Khuri et al. in our SH, resulting in an algorithm in which the current state in the search is represented by a bitstring. A neighbor of the current state is chosen by selecting a single bit uniformly at random and inverting it. The initial state is a bitstring chosen uniformly at random from $\{0,1\}^n$.

Khuri et al. consider three problems: two randomly generated graphs on 20 vertices, and an explicitly constructed graph on 100 vertices. As they perform their first set of experiments on two fixed instances of 20 vertices, rather than a distribution of randomly generated graphs, so that we cannot precisely duplicate their experiments, and as these graphs are somewhat small in any case, we confine our investigation to the 100 vertex graph. The authors construct this graph as a chain of four-vertex components in such a way that it is easy to demonstrate a unique, optimal cut of weight 1077. For details, see [20].

Khuri et al. run a GA with a population size of 50 for 1000 iterations, so that their GA executes a total 50,000 function evaluations. They perform 100 experiments. We compare their results with those of our SH algorithm run for 50,000 iterations, likewise over 100 experiments.

The following table represents a histogram of the maximal fitnesses attained by each algorithm over the above experiments:

| $f$ | $GA$ | $SH$ |
|---|---|---|
| **1077** | 6 | 60 |
| **1055** | 12 | 39 |
| **1033** | 30 | 1 |
| **1011** | 35 | |
| **989** | 12 | |
| **967** | 3 | |
| **945** | 1 | |

The average fitness achieved by the GA was 1022.66 with a standard deviation of 26.12, while the average fitness achieved by the SH algorithm was 1067.98 with a standard deviation of 11.26. On this problem instance, the performance of the basic SH algorithm was markedly superior to that of the GA.

### 3.3 Genetic Programming

"Genetic programming" (GP) is a method of enabling a genetic algorithm to search a potentially infinite space of computer programs, rather than a space of fixed-length solutions to a combinatorial optimization problem. These programs take the form of Lisp symbolic expressions, called *S-expressions*. The idea of applying GAs to S-expressions rather than combinatorial structures is due originally to Fujiki and Dickinson [12] [11], and was brought to prominence through the work of Koza [22]. The S-expressions in GP correspond to programs which a user seeks to adapt to perform some pre-specified task. The fitness of an S-expression may therefore be evaluated in terms of how effectively its corresponding program performs this task. Details on GP, an increasingly common GA application, and on the 11-multiplexer problem which we address in this section, may be found, for example, in [22] [21] [23].

The boolean 11-multiplexer problem entails the generation of a program to perform the following task. A set of 11 inputs is provided, with labels $a_0, a_1, a_2, d_0, d_1, \ldots, d_7$, where $a$ stands for "address" and $d$ for "data". Each input takes the value 0 or 1. The task is to output the value $d_m$, where $m = a_0 + 2a_1 + 4a_2$. In other words, for any 11-bit string, the input to the "address" variables is to be interpreted as an index to a specific "data" variable, which the program then yields as output. For example, on input $a_1 = 1, a_0 = a_2 = 0$, and $d_2 = 1, d_0 = d_1 = d_3 = \ldots = d_7 = 0$, a correct program will output a '1', since the input to the 'a' variables specifies address 2, and variable $d_2$ is given input 1.

**The GA** Koza's GP involves the use of a GA to generate an S-expression corresponding to a correct 11-multiplexer program. An S-expression comprises a tree of LISP *operators* and *operands*, operands being the set of data to be processed — the leaves of the tree — and operators being the functions applied to these data and internally in the tree. The nature of the operators and operands will depend on the problem at hand, since different problems will involve different sets of inputs and will require different functions to be applied to these inputs. For the 11-multiplexer problem in particular, where the goal is to create a specific boolean function, the operands are the input bits $a_0, a_1, a_2, d_0, d_1, \ldots, d_7$, and the operators are AND, OR, NOT, and IF. These operators behave as expected: the subtree (AND $a_1$ $a_2$), for instance, yields the value $a_1 \wedge a_2$. The subtree (IF $a_1$ $d_4$ $d_3$) yields the value $d_4$ if $a_1 = 0$ and $d_3$ if $a_1 = 1$ (and thus can be regarded as a "3-multiplexer"). NOT and OR work similarly. An S-expression constitutes a tree of such operators, with operands at the leaves. Given an assignment to the operands, this tree is evaluated from bottom to top in the obvious way, yielding a 0 or 1 output at the root.

Koza makes use of a "mating" operation in his GA which swaps subexpressions between two such S-expressions. For instance, if the subexpression (NOT $d_2$) were mated with the subexpression (IF $a_0$ $d_7$ (OR $d_3$ $a_1$)), one pair of possible resulting subexpressions would be (NOT (OR $d_3$ $a_1$)) and (IF $a_0$ $d_7$ $d_2$). The subexpressions to be swapped are chosen uniformly at random from the set of all subexpressions in the tree. In every iteration of the GA, fitness-proportionate crossover, according to the above scheme, is applied to 90% of the population, while a somewhat involved type of selection known as "overselection" is applied to the remaining 10%. The fitness of an S-expression is computed by evaluating it on all 2048 possible inputs, and counting the number of correct outputs. Koza does not employ a mutation operator in his GA.

**The SH Algorithm** The hillclimbing algorithm we implemented for this problem was somewhat more sophisticated than for the other three problems described in this paper. In choosing a neighboring state in a way natural to the problem, we arrive at an algorithm which does not make transitions in the neighborhood graph with uniform probability. Another peculiarity of this problem is the fact that the search space – which comprises the set of all legal S-expressions – is infinite.

The initial state in the SH algorithm is an S-expression consisting of a single operand chosen uniformly at random from $\{a_0, a_1, a_2, d_0, \ldots, d_7\}$. A transition in the search space involves the random replacement of an arbitrary node in the S-expression. In particular, to select a neighboring state, we chose a node uniformly at random from the current tree and replace it with a node selected randomly from the set of all possible operands and operators. With probability $\frac{1}{2}$ the replacement node is drawn uniformly at random from the set of operands $\{a_0, a_1, a_2, d_0, \ldots, d_7\}$, otherwise it is drawn uniformly at random from the set of operators, {AND, OR, NOT, IF}. In modifying the nodes of the S-expression in this way, we may change the number of inputs they require. By changing an AND node to a NOT node, for instance, we reduce the number of inputs taken by the node from 2 to 1. In order to accommodate such changes, we do the following. Where a replacement reduces the number of inputs taken by a node, we remove the required number of children from that node uniformly at random. Where, on the other hand, a replacement increases the number of inputs taken by a node, we add the required number of children chosen uniformly at random from the set of operands $\{a_0, a_1, a_2, d_0, \ldots, d_7\}$.

**Experimental Results** In the implementation described in [23], Koza initializes the GA with a pool of 4000 expressions. He observes over 54 experiments that the algorithm has a 28% chance of producing a correct S-expression by the tenth generation (i.e., after 40,000 function evaluations), a 78% chance by the fifteenth generation (after 60,000 function evaluations), and a 90% chance by the twentieth generation (after 80,000 function evaluations). In [21], where Koza performs a series of 21 runs with a slightly different selection scheme from the one described above, he finds that the average number of function evaluations required to find a correct S-expression is 46,667. Here the probability of producing a correct expression after 40,000 function evaluations is about 60% (according to a reading of the graph presented in the paper) and the probability by 60,000 function evaluations is 100%.

In 100 runs of the SH algorithm, we find that the probability of producing a correct S-expression in fewer than 20,000 function evaluations is 61%. The probability that a correct S-expression is found in fewer than 40,000 function evaluations is 98%, in fewer than 60,000 function evaluations, 99%, and in fewer than 80,000 function evaluations, 100%. The average time required to find a correct S-expression was 19,234.90 function evaluations, with a standard deviation of 5179.45. The minimum time to find a correct expression in these runs was 3733, and the maximum, 73,651.

The average number of nodes in the correct S-expression found by the SH algorithm was 88.14; the low was 42, the high, 242, and the standard deviation, 29.16. Given that there are four possible operands (and even more possible operators), it is easy to see that the number of S-expressions of 242 or fewer nodes is greater than $4^{242} \approx 10^{144}$. Thus the portion of the search space which the algorithm explores in practice appears to be of non-trivial size.

The following is a histogram of the number of function evaluations – rounded to the nearest thousand – which SH performed in these 100 experiments in order to achieve a correct S-expression.

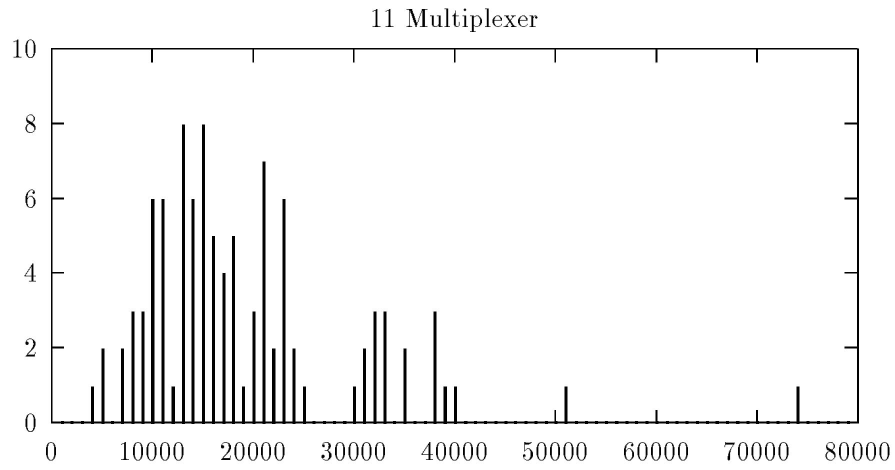


Figure 1. SH on the 11-Multiplexer Problem. Function Evaluations to Achieve Optimum.

**Remark.**

It is interesting to note – perhaps partly in explanation of the SH algorithm's success on this problem – that the SH algorithm formulated here defines a neighborhood structure in which there are *no strict local minima*. More precisely, from any point in the search space, the graph defining the neighborhood structure contains a path to some optimal solution such that every transition in the path leads to a state with an equal or greater fitness. This property holds not only for the 11-multiplexer problem, but for any problem which involves the realization of a specific boolean formula.

To see this, it suffices to observe that the output yielded by a given leaf $q$ can be "corrected" for some arbitrary input $X$ to the operands of the tree. In other words, there is a sequence of transitions in the search space which replaces an arbitrary leaf $q$ by an expression $E$ with the following property: on all inputs but $X$, $E$ yields the same output as $q$; on input $X$, $E$ yields an output complementary to that of $q$. For this corrective sequence of transitions to occur with positive probability, it must of course never result in a decrease in the fitness of the tree. Given this criterion, observe that the following sequence achieves the desired correction:

```
          IF                IF              IF
q  -->  /  |  \  ...  -->  /  |  \   -->   /  |  \     =  E
       q   q   q         E'  q   q       E'  q  NOT
                                                  |
                                                  q
```

where $E'$ is some subexpression which yields a 1 on input $X$ and a 0 otherwise.

In consequence of this observation, a simple inductive proof suffices to show that the search space contains no strict local minima. $\square$

### 3.4 MDAP

The authors of [30] formulate an NP-complete document allocation problem which they refer to as MDAP (Multiprocessor Document Allocation Problem). Their efforts to solve this problem using a GA, which they show to be superior to certain more traditional methods, have received considerable attention. In this section we will perform two experiments comparing the performance of a GA developed in that paper to an SH algorithm for MDAP. We will conduct the first of these experiments on a suite of problems drawn from [30], and the second on a new suite of problems developed for the purposes of the current paper.

The inputs to the MDAP problem are a multiprocessor architecture, a collection of documents, and a collection of subsets of these documents, or *clusters*. The problem is to allocate the documents to the processors so as to minimize the interprocessor distances between documents in the same cluster. This minimization of distances within clusters is a heuristic meant to produce a low retrieval time for documents in the same cluster.

More formally, an instance of MDAP is specified by the following parameters:

- A set $X = \{x_1, x_2, \ldots, x_n\}$ of processors.
- An architecture on these processors, specified by interprocessor distances $X_{ij}$ for $1 \leq i < j \leq n$.
- A set $D = \{d_1, d_2, \ldots, d_m\}$ of documents.
- A set $\{C_1, C_2, \ldots, C_r\}$ of clusters, where each cluster $C_i$ comprises a set of documents, $\{d^i_1, d^i_2, \ldots, d^i_{|C_i|}\}$.

An allocation $A$ is a mapping $D \rightarrow X$, that is, a placement of documents on processors. As a heuristic for ensuring an evenly distributed workload, an allocation is required to distribute the documents so that every processor receives either $\lceil |D|/|X| \rceil$ or $\lfloor |D|/|X| \rfloor$

documents. The quantity that determines a cluster's retrieval time is its *radius*, which is the maximum interprocessor distance of any two elements in the cluster. MDAP seeks to minimize the sum of the radii of the clusters. More formally, if $R(C_i)$ is the radius of cluster $C_i$, what is sought is a function $A : D \to X$ such that:

$$\sum_{i=1}^{r} R(C_i)$$

is minimized. Significant overlap among clusters can result in complex tradeoffs in the minimization of cluster radii. MDAP is NP-complete for general architectures [30].

**The GA** In the GA developed in [30] specifically for this problem, feasible solutions take the form of permutations. The permutation $\pi : \{1 \ldots n\} \to \{1 \ldots n\}$ specifies the document allocation $A(d_i) = \pi(i) \pmod{|X|}$. This scheme ensures an even distribution of documents across processors. The GA attempts to minimize a fitness function $f$, where for a permutation $\pi$, $f(\pi)$ is defined to be the sum of the cluster radii specified by $\pi$.

This encoding necessitated the development of special mating and mutation operators on permutations. The mating operator applied to a pair $(\alpha, \beta)$ of permutations does the following. A position $i$ is chosen uniformly at random from $\{1, \ldots, n\}$, designating a transposition $\tau = (\alpha(i)\ \beta(i))$. This transposition is applied to yield new children $\alpha' = \alpha\tau$ and $\beta' = \beta\tau$. Suppose, for example, that $\alpha =< 2, 1, 3, 4 >$ and $\beta =< 4, 1, 2, 3 >$, and that position $i = 4$ is chosen. Since $\alpha(4) = 4$ and $\beta(4) = 3$, the mating operator will compose $\alpha =< 2, 1, 3, 4 >$ with the transposition (34), yielding $\alpha' =< 2, 1, 4, 3 >$ and $\beta =< 3, 1, 2, 4 >$ with transposition (34), yielding $\beta' =< 4, 1, 2, 3 >$. This is the basic operation. The full blown mating operator in fact picks two positions $j \leq k$ uniformly at random from $\{1, \ldots, n\}$ and performs the above mating operation on position $j$, then $j + 1$, and so forth, through $k$. Those familiar with GAs will observe that this mating operator is slightly unusual in that there is no exchange of material between solutions.

The mutation operator as applied to permutation $\alpha$ simply selects two elements $\alpha(i)$ and $\alpha(j)$ (not necessarily distinct) uniformly at random from among the elements in $\alpha$ and transposes them.

For the purposes of the experiments to come in the latter half of this section, we attempted to recode the GA in [30] according to the description presented in the paper. Our results, however, were consistently inferior to those given by the authors of that article. This situation was aggravated by the fact that their code was unavailable for distribution due to a computer crash [10]. We found, however, that by implementing the mating and mutation operators in [30] in the context of a different formulation of the GA we were able to achieve results superior to those presented by the authors.

Our GA includes, in order, the following phases: evaluation, elitist replacement, selection, crossover, and mutation. In the evaluation phase, the fitnesses of all members of the population are computed. Elitist replacement substitutes the fittest permutation from the evaluation phase of the previous iteration for the least fit permutation in the current population (except, of course, in the first iteration, in which there is no replacement). The GA proposed in [30] employs unscaled fitness-proportionate selection. The performance of this variety of selection on function optimization tasks is often found to be somewhat poor, so we chose instead to use binary stochastic tournament selection [15]. In this type of selection, $P$ pairs, where $P$ is the size of the population, are selected uniformly at random with replacement from the population. A new population of size $P$ is constituted by selecting the fitter permutation from each of these pairs (with ties broken randomly). The crossover step in our GA selects $\frac{P}{2}$ pairs uniformly at random without replacement from the population and applies the mating operator to each of these pairs independently with probability 0.6. For this probability, referred to generally as the *crossover rate*, a value of 0.6 is fairly standard in the literature. It is, for instance, the default value proposed in Greffenstette's GENESIS software package [16]. In accordance with results in [3] and [26] regarding the optimal *mutation rate*, we set this parameter to $\frac{1}{n}$. More precisely, the number of mutations performed on a given permutation in a single iteration is binomial with parameter $p = \frac{1}{n}$. The population in our GA is initialized by selecting every individual uniformly at random from $S_n$.

**The SH Algorithm** As in the GA above, the search space $S$ for the SH algorithm we developed was $S_n$, the space of all permutations on $n$ elements. The initial state $\sigma$ is chosen uniformly at random from $S_n$. A neighbor is selected by transposing two elements chosen uniformly at random from the permutation. In other words, to select a neighbor $\tau$, a single mutation of the sort employed in the GA described above is applied.

Four architectures are considered in [30]: a 1x16 mesh, a 2x8 mesh, a 4x4 mesh, and a hypercube of 16 processors. The authors consider two types of document distribution, an "even" distribution and a "(64, 8, 25, 50)" distribution. In the "even" distribution, 64 documents are assigned to 8 clusters, each of size 8. Since each processor must be assigned an equal number of documents, an allocation in this instance will map 4 documents to every processor. In this "even" distribution of documents, the clusters are disjoint. On the one hand, this implies that the solution to the problem is trivial: if each cluster is assigned to two adjacent processors, the radius for each cluster will be equal to 1, which is the smallest we can achieve, since every cluster contains 8 documents and therefore must be placed on at least two processors. On the other hand, this means that we know the minimal cluster radius sum for this problem: for all of the architectures presented here, it is equal to 8. In consequence, we can gauge the performance of the two algorithms here in an absolute sense. As the other distribution considered in their paper, the "(64, 8, 25, 50)" – in which half of the documents occupy one quarter of the clusters – appears to have a similar triviality, we consider it sufficient just to examine the "even" distribution.

The authors of [30] tabulate their average results for a GA on a population of size 30 executed for 1000 iterations. We present their results, indicated by "ga", with those obtained by our own GA, indicated by "GA", and with those of our SH algorithm, indicated by "SH". The results for the GA in [30] are the average results over 5 trials [10]. The results for our GA are the average results over 100 runs on a population size of 30 for 1000 iterations. The results for the SH algorithm are the average of 100 runs, each executed for 30,000 iterations. For experiments with our GA and SH, we also present the standard deviation on the final fitnesses obtained in our experiments (indicated by "SD"), the highest and lowest final fitnesses, and the number of times that the optimum fitness was achieved ("#Opt.").

For each architecture, we generate a single instance at random, and apply both algorithms to that one instance. Note that by merit of its having disjoint clusters, however, all instances are identical up to relabeling, so that the specific choice of problem instance has no bearing on the performance of the GA or the SH algorithm.

| | **1x16 Mesh** | | | **2x8 Mesh** | | | **4x4 Mesh** | | | **Hypercube** | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | ga | GA | SH | ga | GA | SH | ga | GA | SH | ga | GA | SH |
| Mean | 29 | 18.21 | 8.78 | 23 | 16.85 | 11.95 | 19 | 17.49 | 12.76 | 23 | 17.65 | 15.58 |
| SD | | 2.69 | 1.05 | | 1.24 | 1.81 | | 1.32 | 2.01 | | 0.94 | 0.79 |
| High | | 25 | 12 | | 20 | 16 | | 21 | 16 | | 20 | 17 |
| Low | | 13 | 8 | | 14 | 8 | | 15 | 8 | | 15 | 13 |
| #Opt. | | 0 | 63 | | 0 | 4 | | 0 | 5 | | 0 | 0 |

The following graph represents the performance of the two algorithms on the 1x16 mesh architecture. The graphs for the other three architectures display the same basic form.

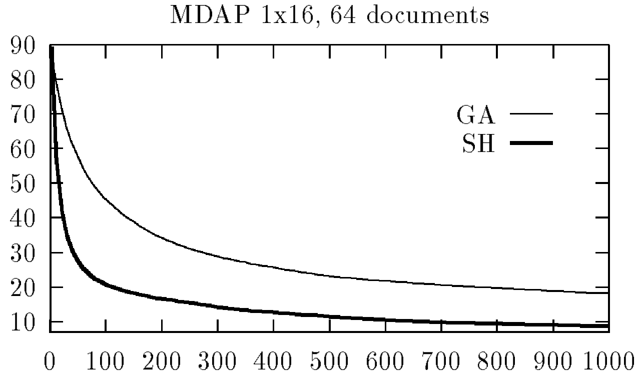


Figure 2. MDAP, 1x16 MESH, Avg. Fitness vs. Number of Iterations

It could be objected that it is the simplicity of the instances of this problem which is responsible for the relative success of the hillclimbing approach. To demonstrate that this is not the case, we modified the suite of problems used in the first set of experiments to create more difficult instances of MDAP. In our second set of experiments, we explore a deeper range of "even" distributions in which there is overlap among clusters, insuring a non-trivial search space. We refer to as an $m \times m$ problem one in which there are $m$ clusters, each containing $m$ documents. In the set of problems we consider here, each cluster is assigned a collection of documents chosen uniformly at random without replacement from the total set of documents to be assigned.

Our second set of experiments includes a 10x10 and a 12x12 problem on a 64 document set, and a 12x12 and a 14x14 problem on a 128 document set. For each of these problems, we consider two 16 processor architectures, namely a 2x8 mesh and a hypercube. We also examine two slightly larger instances, a 18x18 and a 20x20 problem on a 256 document set. We consider each of these two problems on two different 32 processor architectures, a 4x8 mesh and a hypercube.

For our GA, we present the average results obtained over 100 experiments of 1000 iterations on a population size of 100. For the SH algorithm we present the average fitness obtained over 100 runs, each comprising 100,000 iterations. We generate a new random instance of each problem individually for every run.

Results for the 64 document instances are as follows:

| | **2x8 Mesh, 10x10** | | **Hypercube, 10x10** | | **2x8 Mesh, 12x12** | | **Hypercube, 12x12** | |
|---|---|---|---|---|---|---|---|---|
| | GA | SH | GA | SH | GA | SH | GA | SH |
| Mean | 37.49 | 30.31 | 29.02 | 24.65 | 52.75 | 46.20 | 38.11 | 34.57 |
| SD | 1.46 | 1.19 | 0.85 | 0.73 | 1.72 | 1.09 | 0.93 | 0.88 |
| High | 42 | 34 | 31 | 26 | 58 | 49 | 40 | 36 |
| Low | 34 | 28 | 27 | 23 | 49 | 44 | 36 | 33 |

For the 128 document instances:

| | **2x8 Mesh, 12x12** | | **Hypercube, 12x12** | | **2x8 Mesh, 14x14** | | **Hypercube, 14x14** | |
|---|---|---|---|---|---|---|---|---|
| | GA | SH | GA | SH | GA | SH | GA | SH |
| Mean | 43.32 | 31.27 | 35.12 | 31.04 | 59.72 | 46.49 | 45.11 | 40.41 |
| SD | 1.85 | 0.98 | 0.89 | 1.26 | 1.82 | 1.33 | 1.14 | 1.02 |
| High | 49 | 34 | 37 | 34 | 64 | 51 | 48 | 42 |
| Low | 38 | 29 | 33 | 28 | 55 | 42 | 43 | 38 |

And for the 256 document instances:

| | **4x8 Mesh, 18x18** | | **Hypercube, 18x18** | | **4x8 Mesh, 20x20** | | **Hypercube, 20x20** | |
|---|---|---|---|---|---|---|---|---|
| | GA | SH | GA | SH | GA | SH | GA | SH |
| Mean | 101.09 | 79.64 | 73.96 | 71.78 | 123.01 | 97.44 | 85.93 | 80.86 |
| SD | 2.58 | 1.55 | 1.01 | 0.46 | 2.89 | 1.83 | 1.32 | 0.44 |
| High | 108 | 82 | 77 | 72 | 130 | 101 | 89 | 81 |
| Low | 95 | 76 | 72 | 70 | 116 | 93 | 83 | 79 |

It will be observed that here again, in the case of these more difficult instances of MDAP, the SH algorithm outperformed the GA on all problem sizes.

### 3.5 Jobshop

In this section, we formulate an SH algorithm for the jobshop problem. Rather than simply constructing this algorithm based on the mutation operator in the GA with which we compare our results – as we did in the case of the maximum cut and MDAP problems – we consider the combinatorial structure of the problem afresh. This leads us to conceive a new neighborhood structure for jobshop which we shall employ in the following section to construct a new, more effective GA for the problem.

The jobshop problem is widely studied in the field of management science. It is a notoriously difficult NP-complete problem [13] that is hard to solve even for small instances. A great deal of effort over the course of thirty years has gone into finding efficient approximation algorithms for it. See, for example, [4, 5, 7, 24, 6, 28, 9, 27].

In this problem, a collection of $J$ jobs are to be scheduled on $M$ machines (or processors), each of which can process only one task at a time. Each job is a list of $M$ tasks which must be performed in order. Each task must be performed on a specific machine, and no two tasks in a given job are assigned to the same machine. Every task has a fixed (integer) processing time. The problem is to schedule the jobs on the machines so that all jobs are completed in the shortest overall time. This time is referred to as the *makespan*.

Three instances formulated in [27] constitute a standard benchmark for this problem: a 6 job, 6 machine instance, a 10 job, 10 machine instance, and a 20 job, 5 machine instance. The 6x6 instance is now known to have an optimal makespan of 55. This is very easy to achieve. While the optimum value for the 10x10 problem is known to be 930, this is a difficult problem which remained unsolved for over 20 years [2]. A great deal of research has also been invested in the similarly challenging 20x5 problem, for which an optimal value of 1165 has been achieved, and a lower bound of 1164 [6].

A number of papers have considered the application of GAs to scheduling problems. In particular, Nakano and Yamada [28], Davidor et al. [7], and Fang et al. [9] have described

GAs designed to address the three benchmark instances for the jobshop problem. We compare our results with those obtained in Fang et al., one of the more recent of these articles.

**The GA** Fang et al. encode a jobshop schedule in the form of a string of integers, to which their GA applies a conventional crossover operator. This string contains $JM$ integers $a_1, a_2, \ldots, a_{JM}$ in the range $1..J$. A circular list $C$ of jobs, initialized to $(1, 2, \ldots, J)$ is maintained. For $i = 1, 2, \ldots, JM$, the first uncompleted task in the $(a_i \bmod |C|)^{th}$ job in $C$ is scheduled in the earliest plausible timeslot. A *plausible* timeslot is one which comes after the last scheduled task in the current job, and which is at least as long as the processing time of the task to be scheduled. When a job is complete, it is removed from $C$. Fang et al. also develop a highly specialized GA for this problem in which they use a scheme of increasing mutation rates and a technique known as GVOT (Gene-Variance based Operator Targeting). For the details see [9].

**The SH Algorithm** In our SH algorithm for this problem, a schedule is encoded in the form of an ordering $\sigma_1, \sigma_2, \ldots, \sigma_{JM}$ of $JM$ markers. These markers have colors associated with them: there are exactly $M$ markers of each color of $1, \ldots, J$. To construct a schedule, $\sigma$ is read from left to right. Whenever a marker with color $k$ is encountered, the next uncompleted task in job $k$ is scheduled in the earliest plausible timeslot. Since there are exactly $M$ markers of each color, and since every job contains exactly $M$ tasks, this decoding of $\sigma$ yields a complete schedule. Observe that since markers of the same color are interchangeable, many different ordering $\sigma$ will correspond to the same scheduling of tasks.

To generate a neighbor of $\sigma$ in this algorithm, a marker $\sigma_i$ is selected uniformly at random and moved to a new position $j$ chosen uniformly at random. To achieve this, it is necessary to shift the subsequence of markers between $\sigma_i$ and $\sigma_j$ (including $\sigma_j$) one position in the appropriate direction. If $i < j$, then $\sigma_{i+1}, \sigma_{i+2}, \ldots, \sigma_j$ are shifted one position to the left in $\sigma$. If $i > j$, then $\sigma_j, \sigma_{j+1}, \ldots, \sigma_{i-1}$ are shifted one position to the right. (If $i = j$, then the generated neighbor is of course identical to $\sigma$.)

Suppose, for example, that $J = 2$ and $M = 3$. One possible ordering $\sigma$ corresponds to the sequence of colors 111222. (Note that in the above formulation, it is only the colors of the markers that are significant.) If we choose to move $\sigma_1$ (here, the first marker of color 1) to position 6, then we obtain the sequence 112221.

Fang et al. consider the makespan achieved after 300 iterations of their GVOT-based GA on a population of size 500. We compare this with an SH for which each experiment involves 30,000 iterations repeated 5 times; we take the result of the best repetition. In both cases therefore, a single execution of the algorithm involves a total of 150,000 function evaluations. Note that multiple repetitions of the SH algorithm, as remarked in Section 2, proved important to its performance. We chose the number of repetitions in the SH algorithm arbitrarily; presumably, it could be improved with tuning.

Fang et al. present their average results over 10 trials, but do not indicate how they obtain their "best". We present the statistics resulting from 100 executions of the SH algorithm. Here, as in all subsequent tables, "Mean" indicates the mean result obtained in these 100 runs, while "High" and "Low" indicate the highest and lowest quantities obtained. "SD" indicates the standard deviation of the final fitnesses achieved in these 100 runs. The "Best Known" results are the best upper bounds for these problem instances available in the literature.

| | **10x10 Jobshop** | | **20x5 Jobshop** | |
|---|---|---|---|---|
| | GA | SH | GA | SH |
| Mean | 977 | 965.64 | 1215 | 1204.89 |
| SD | | 10.56 | | 12.92 |
| High | | 996 | | 1241 |
| Low | 949 | 949 | 1189 | 1183 |
| Best Known | *930* | | *1165* | |

As can be seen from the above table, the performance of the SH algorithm appears to be as good as or superior to that of the GA.

### 3.6 A New Jobshop GA

In this section, we reconsider the jobshop problem in an attempt to formulate a new GA encoding. We use the same encoding as in the SH algorithm described above: $\sigma$ is an ordering $\sigma_1, \sigma_2, \ldots, \sigma_{JM}$ of the $JM$ markers, which can be used to construct a schedule as before. We treated markers of the same color as effectively equivalent in the SH algorithm. Now, however, the label of a marker (a unique integer in $\{1, \ldots, JM\}$) will play a role.

The basic step in the crossover operator for this GA as applied to a pair $(\sigma, \tau)$ of orderings is as follows. A label $i$ is chosen uniformly at random from $\{1, 2, \ldots, JM\}$. In $\sigma$, the marker with label $i$ is moved to the position occupied by $i$ in $\tau$; conversely, the marker with label $i$ in $\tau$ is moved to the position occupied by that marker in $\sigma$. In both cases, the necessary shifting is performed as before. Hence the idea is to move a single marker in $\sigma$ (and in $\tau$) to a new position as in the SH algorithm; instead of moving the marker to a random position, though, we move it to the position occupied by that marker in $\tau$ (and $\sigma$, respectively). The full crossover operator picks two labels $j \leq k$ uniformly at random from $\{1, 2, \ldots, JM\}$, and performs this basic operation first for label $j$, then $j+1$, and so forth, through $k$. (By analogy with the GA above for MDAP.) The mutation operator in our GA performs exactly the same operation as that used to generate a neighbor in the SH algorithm. A marker $\sigma_i$ is chosen uniformly at random and moved to a new position $j$, chosen uniformly at random. The usual shifting operation is then performed. Observe how closely the crossover and mutation operators in this GA for the jobshop problem are based on those in the corresponding SH algorithm. In all other details, the GA here is identical to that implemented for the MDAP problem above.

We execute this GA for 300 iterations on a population of size 500. Results of 100 experiments performed with this GA are indicated in the following table by "new GA". For comparison, we again give the results obtained by the GA of Fang et al. and the SH algorithm described in this paper.

| | **10x10 Jobshop** | | | **20x5 Jobshop** | | |
|---|---|---|---|---|---|---|
| | new GA | GA | SH | new GA | GA | SH |
| Mean | 956.22 | 977 | 965.64 | 1193.21 | 1215 | 1204.89 |
| SD | 8.69 | | 10.56 | 7.38 | | 12.92 |
| High | 976 | | 996 | 1211 | | 1241 |
| Low | 937 | 949 | 949 | 1174 | 1189 | 1183 |
| Best Known | *930* | | | *1165* | | |

With this new encoding inspired by our SH algorithm, we see that the GA was able to outperform the SH developed for this problem. It is not possible to compare the effectiveness of this GA encoding directly to the encoding proposed in Fang et al. or Davidor et al., because their GAs differed from our own: Fang et al. employ a specialized crossover operator and Davidor et al., a "steady-state" GA. The superiority of the experimental results presented here over those previously published, though, would seem to suggest that the above encoding is at least as effective as these others – even allowing for differences in the respective powers of the GAs involved. More importantly, the encoding presented here was simpler than previous ones, an advantage offered by our consideration of a straightforward optimization approach before we developed our GA encoding.

## 4 Conclusion

As black-box algorithms, GAs are principally of interest in solving problems whose combinatorial structure is not understood well enough for more direct, problem-specific techniques to be applied. As we have seen in the case of the four problems presented in this paper, stochastic hillclimbing can offer a useful gauge of the performance of the GA. In some cases it shows that a GA-based approach may not be competitive with simpler methods; at others it offers insight into possible design decisions for the GA such as the choice of encoding and the formulation of mating and mutation operators. We again stress that we do not advocate stochastic hillclimbing as a powerful optimization technique in itself. Given its simplicity, there are likely to be substantially more effective optimization heuristics in most cases. Nonetheless, in light of the results presented in this paper, we hope that designers of black-box algorithms will be encouraged to experiment with stochastic hillclimbing in the initial stages of the development of their algorithms.

**Acknowledgments**

We wish to thank Alistair Sinclair for suggesting a simplification to our GA encoding for the jobshop problem and for his comments on drafts of this paper. Thanks also to Dave Corne, Jean-Arcady Meyer, and Hava Siegelmann.

# References

[1] D. Ackley. *A Connectionist Machine for Genetic Hillclimbing.* Kluwer Academic Publishers, 1987.

[2] D. Applegate and W. Cook. A computational study of the job-shop problem. *ORSA Journal of Computing*, 3(2), 1991.

[3] T. Bäck. The interaction of mutation rate, selection, and self-adaptation within a genetic algorithm. In R. Männer and B. Manderick, editors, *Parallel Problem Solving from Nature 2*, pages 85–94. Elsevier, 1992.

[4] E. Balas. Machine sequencing via disjunctive graphs: An implicit enumeration algorithm. *Operations research*, 17:941–957, 1969.

[5] J. Barker and G. McMahon. Scheduling the general jobshop. *Management Science*, 31(5):594–598, 1985.

[6] J. Carlier and E. Pinson. An algorithm for solving the jobshop problem. *Management Science*, 35:(2):164–176, 1989.

[7] Y. Davidor, T. Yamada, and R. Nakano. The ECOlogical framework II: Improving GA performance at virtually zero cost. In Forrest, editor, *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 171–176, San Mateo, CA, 1993. Morgan Kaufmann.

[8] L. Davis. Bit-climbing, representational bias, and test suite design. In Belew and Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 18–23, San Mateo, CA, 1991. Morgan Kaufmann.

[9] H. Fang, P. Ross, and D. Corne. A promising genetic algorithm approach to job-shop scheduling, rescheduling, and open-shop scheduling problems. In Forrest, editor, *Proceedings of the Fifth International Conference on Genetic Algorithms*, San Mateo, CA, 1993. Morgan Kaufmann.

[10] O. Frieder. Personal communication, 1993.

[11] C. Fujiki. An evaluation of Holland's genetic operators applied to a program generator. Master's thesis, University of Idaho, 1986.

[12] C. Fujiki and J. Dickinson. Using the genetic algorithm to generate Lisp source code to solve the prisoner's dilemma. In J. Grefenstette, editor, *Proceedings of the Second International Conference on Genetic Algorithms*, pages 236–240, San Mateo, CA, 1989. Morgan Kaufmann.

[13] M. Garey and D. Johnson. *Computers and Intractability.* W.H. Freeman and Co., 1979.

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

[15] D. Goldberg and K. Deb. A comparative analysis of selection schemes used in genetic algorithms. In *Foundations of Genetic Algorithms 2*, pages 69–93, San Mateo, CA, 1991. Morgan Kaufmann.

[16] J. Grenfenstette. *A User's Guide to GENESIS.* Navy Center for Applied Research in Artificial Intelligence, 1987.

[17] D. Johnson, C. Aragon, L. McGeoch, and C. Schevon. Optimization by simulated annealing: An experimental evaluation; part I, graph partitioning. *Operations Research*, 37(6), 1989.

[18] D. Johnson, C. Aragon, L. McGeoch, and C. Schevon. Optimization by simulated annealing: An experimental evaluation; part II, graph coloring and number partitioning. *Operations Research*, 39(3), 1991.

[19] K. De Jong. *An Analysis of the Behavior of a Class of Genetic Adaptive Systems.* PhD thesis, University of Michigan, 1975.

[20] S. Khuri, T. Bäck, and J. Heitkötter. An evolutionary approach to combinatorial optimization problems. In *Proceedings of CSC 1994*, 1994.

[21] J. Koza. *Foundations of Genetic Algorithms*, chapter A Hierarchical Approach to Learning the Boolean Multiplexer Function, pages 171–192. Morgan Kaufmann, San Mateo, CA, 1991.

[22] J. Koza. *Genetic Programming.* MIT Press, Cambridge, Massachusetts, 1991.

[23] J. Koza. The genetic programming paradigm: Breeding computer programs. In Branko Souček and the IRIS Group, editors, *Dynamic, Genetic, and Chaotic Programming*, pages 203–221. John Wiley and Sons, Inc., 1992.

[24] G. McMahon and M. Florian. On scheduling with ready times and due dates to minimize maximum lateness. *Operations research*, 23(3):475–482, 1975.

[25] M. Mitchell, J. Holland, and S. Forrest. When will a genetic algorithm outperform hill-climbing? In J.D. Cowen, G. Tesauro, and J. Alspector, editors, *Advances in Neural Information Processing Systems 6*, San Mateo, CA, 1994. Morgan Kaufmann.

[26] H. Mühlenbein. How genetic algorithms really work: I. mutation and hillclimbing. In R. Männer and B. Manderick, editors, *Parallel Problem Solving from Nature 2*, pages 15–25. Elsevier, 1992.

[27] J. Muth and G. Thompson. *Industrial Scheduling.* Prentice Hall, Englewood Cliffs, New Jersey, 1963.

[28] R. Nakano and T. Yamada. Conventional genetic algorithm for job shop problems. In Belew and Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 474–479, San Mateo, CA, 1991. Morgan Kaufmann.

[29] C. Shaefer and S. Smith. The Argot strategy II – combinatorial optimizations. Technical Report RL90-1, Thinking Machines, 1990.

[30] H. Siegelmann and O. Frieder. Document allocation in multiprocessor information retrieval systems. In N. Adam and N. Bhargava, editors, *Lecture note series in Computer Science: Advanced Database Systems.* Springer Verlag, 1994.

[31] S. Wilson. GA-easy does not imply steepest-ascent optimizable. In Belew and Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 85–89, San Mateo, CA, 1991. Morgan Kaufmann.