Genetic Placement[†]

James P. Cohoon and William D. Paris
Department of Computer Science
University of Virginia

Computer Science Report No. TR–86-14
Revised May 28, 1987

† This report appears in the November, 1987 issue of *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems.*

GENETIC PLACEMENT

James P. Cohoon and William D. Paris[†]

University of Virginia
Department of Computer Science
Charlottesville, Virginia 22903

KEYWORDS

Placement, Genetic Algorithms, VLSI, Physical Design

ABSTRACT

A placement algorithm, *Genie*, is presented for the assignment of modules to locations on chips. *Genie* is an adaptation of the genetic algorithm technique that has traditionally been a tool of the artificial intelligence community. The technique is a paradigm for examining a state space. It produces its solutions through the simultaneous consideration and manipulation of a set of possible solutions. The manipulations resemble the mechanics of natural evolution. For example, solutions are "mated" to produce "offspring" solutions. *Genie* has been extensively run on a series of small test instances. Its solutions were observed to be good and in several cases optimal.

† William Paris is currently with Syntek Systems of Bethesda, MD.

This report appears in the November, 1987 issue of *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*.

His genie led him into the pleasant paths.
— Anthony Wood, 1662

1. INTRODUCTION

The layout problem is a principal problem in the design of VLSI chips. Because of its complexity, it is often decomposed into several distinct subproblems:

1) Chip Planning,
2) Partitioning,
3) Placement,
4) Routing.

This paper investigates the *placement problem*—the assignment of circuit elements to locations on the chip. The input to our variant of the placement problem is a set of m circuit elements or *modules*, $M=\{e_1,\ldots,e_m\}$, and a set of n signals or *nets*, $N=\{s_1,\ldots,s_n\}$, where a net is a set of modules to be interconnected. We are also given as input a set of l chip locations or *slots*, $L=\{c_1,\ldots,c_l\}$, where $l \geq m$. The slots are organized as a matrix with r rows and c columns. The objective is to optimally assign each module to its own slot while satisfying electrical constraints, where optimality is measured in terms of the expected routability of the placement. Two components common to many routability measures are estimates of the amount of wire congestion, and the amount of wire required to route all interconnections. Minimizing the expected wire congestion is important as a feasible wiring is usually found more readily with less congestion; minimizing the expected amount of wire is important as the circuit's signal propagation rate is typically inversely proportional to the amount of wire.

Because of its importance, the placement problem has received considerable attention and many diverse solution strategies have been proposed: partitioning [BREU77, DUNL85], quadratic assignment [HANA72], force-directed [QUIN75], resistive network [CHEN84], and simulated annealing [KIRK83].

The contribution presented here is a new placement algorithm, *Genie*, which is based on the genetic algorithm approach [HOLL75]. This approach—like simulated annealing—is a paradigm for examining a state-space. It produces good solutions through simultaneous consideration and manipulation of a set of possible solutions. But unlike simulated annealing, which has been applied primarily to combinatorial optimization problems during its five years of use, most previous applications of the genetic algorithm technique have been to artificial intelligence problems [GREF85b]. Based on our preliminary experiments, this new use of genetic algorithms appears to be a promising method of solving placement problems.

2. GENETIC ALGORITHM PRELIMINARIES

Genetic algorithms represent and transform solutions from a problem's state space Π in a way that resembles the mechanics of natural evolution. As a result, much of the terminology is drawn from biology and evolution. The subspace P of Π currently being examined is referred to as the *population.* The population is composed of *strings*, where a string is an encoding ξ of a solution for the problem. A string is a concatenation of symbols or *alleles*, where each allele is an element of an alphabet Σ. Each string is assigned a *score* (positive real number) through a function $\sigma\colon \Pi \rightarrow R^+$. Without loss of generality, we assume that the objective function seeks a global minimum. Hence, string x is preferred to string y if $\sigma(x) < \sigma(y)$.

Each iteration or *generation*, a fraction K_ψ of P is selected to be *parents* by repeated use of the *choice* function, $\phi\colon 2^\Pi \rightarrow \Pi \times \Pi$, where 2^Π is the power set of Π. New solutions, known as *offspring*, are created by combining pairs of parents in such a way that each parent contributes to the information carried by its associated offspring string. This recombination operation, $\psi\colon \Pi \times \Pi \rightarrow \Pi$, is known as *crossover.* After the crossovers are performed, a selector $\rho\colon 2^\Pi \times 2^\Pi \rightarrow 2^\Pi$ is applied to the previous generation and the offspring to determine which strings *survive* to form the basis of the next generation. The number of strings in this next generation typically equals the number in the previous generation. Each surviving string in the population with probability K_μ undergoes *mutation*, $\mu\colon \Pi \rightarrow \Pi$. This perturbation helps prevent a premature loss of diversity within a population by introducing new strings. Over many generations, better scoring strings tend to predominate in the population while less fit strings tend to *die-off*. Eventually one or more super-fit strings *evolve*. A high-level algorithmic description of a basic genetic algorithm is given in Figure 1.

During algorithmic development and experimental analysis, we analyzed multiple string encodings, initial population constructors, crossover and mutation operators, selection techniques, and scoring functions. We also considered several ways to direct interaction among the population. For example, a genetic algorithm that divided the population into two *castes* was developed and was very quickly abandoned. The results of our development and analysis are presented in the next two sections.

```
1.  $P \leftarrow$ initial population constructed with function $\Xi$
2.  $p \leftarrow |P|$
3.  for $i \leftarrow 1$ to $NUMBER\_GENERATIONS$ do
4.      $Offspring \leftarrow \varnothing$
5.      for $k \leftarrow 1$ to $p \cdot K_\psi$ do
6.          $(x, y) \leftarrow \phi(P)$
7.          $Offspring \leftarrow Offspring \cup \{\psi(x, y)\}$
8.      end for
9.      $P \leftarrow \rho(P, Offspring)$
10.     for each string $x \in P$ do
11.         with probability $K_\mu$ mutate $x$ with $\mu$
12.     end for
13. end for
14. return highest scoring string in $P$
```

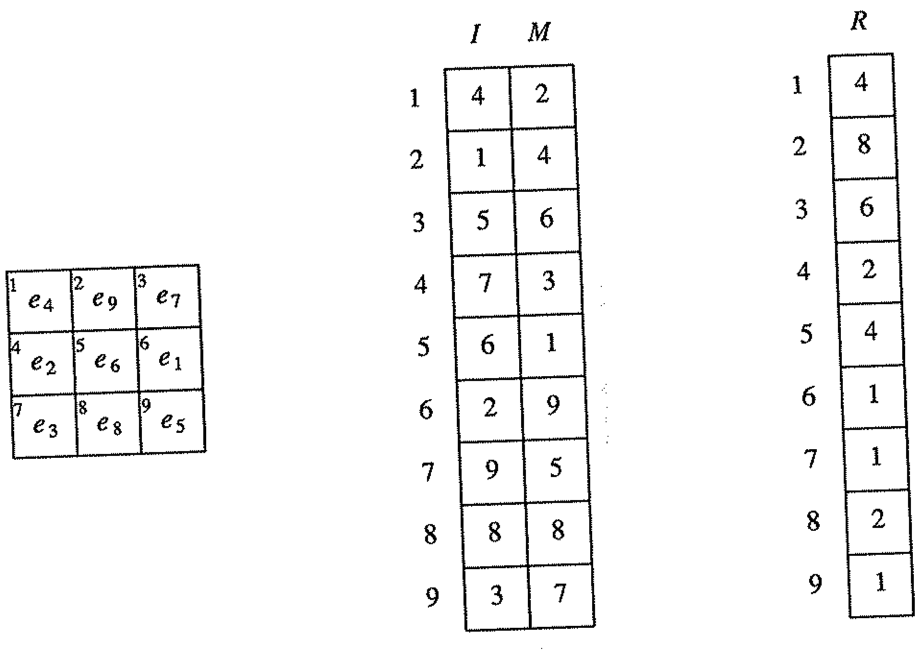
Figure 1 — Genetic Algorithm Paradigm

3. GENIE SPECIFICATION

In its natural setting of biological organisms, evolution is a slow process where new traits are introduced through the random mutation and mixing of genes. From a preliminary investigation into genetic placement, we determined that similar restrictions on operator functionality were not always necessary. Therefore, during *Genie*'s algorithm development we adopted a *directed-evolution* methodology. With this methodology, examination of the state-space is influenced so that solutions with desirable characteristics are obtained more quickly. We avoid a population of inferior local optima by discouraging premature homogeneity through the use of random variates in some of the operators and functions.

3.1. String Encoding ξ

We considered several encodings for the strings, all using l alleles, where l is the number of slots on the chip. Each encoding assumed that the underlying representation of a chip was a matrix organized in row-



(a) Placement (b) ξ_1 (c) ξ_2

Figure 2 — Some String Encodings

major order from top to bottom. Two of these alternative encodings are given in Figures 2.b and 2.c. They describe the module placement that is given in Figure 2.a. In that figure, the *i-th* slot is labeled i in its upper left corner.

The encoding, ξ_1, in Figure 2.b divides each allele into two fields, I_j and M_j, $1 \le j \le l$. Field I_j specifies a location on the chip and field M_j specifies the index of the module in I_j*-th* location. Thus the third allele in Figure 2.b specifies that module e_6 is found in the fifth location. The encoding, ξ_2, in Figure 2.c uses a single field, R_j, for each allele j, $1 \le j \le l$. R_j is used to calculate the module located in the *j-th* location on the chip. Besides the obvious function that directly used R_j as the index of the module in the *j-th* location, we considered a variant of the ordinal function of Grefenstette, Gopal, Rosmaita, and VanGucht [GREF85a]. It works as follows: the module with the R_1*-th* smallest index with respect to all modules is assigned the first location; the module with the R_2*-th* smallest index with respect to the remaining modules is assigned the second location, and so on. Thus as the first allele in the encoding of Figure 2.c contains a 4, module e_4 with the 4-*th* smallest index among modules $\{e_1,e_2,e_3,e_4,e_5,e_6,e_7,e_8,e_9\}$ is assigned to the first slot. As the second allele contains an 8, module e_9 with the 8-*th* smallest index among unassigned modules $\{e_1,e_2,e_3,e_5,e_6,e_7,e_8,e_9\}$ is assigned the second slot. Similarly, as the third allele contains a 6, module e_7 with the 6-*th* smallest index among unassigned modules $\{e_1,e_2,e_3,e_5,e_6,e_7,e_8\}$ is assigned the third slot. Like ξ_1, encoding ξ_2, is particularly suitable for traditional genetic crossover operators that copy a contiguous portion of one parent into the offspring while having the other parent copy over as many other positions as possible, since the resultant solution is automatically a feasible solution. However, neither encoding is amenable to the quick updating of an offspring or mutation's score and neither is suitable for a directed evolutionary crossover or mutation operation. Therefore, we explicitly encoded the solution as a string where the *j-th* allele specified the index of the module in the *j-th* location.

Note that two modules e_s and e_t are *adjacent* if their slots share a common side or corner and are *rectilinearly adjacent* if their slots share a common side. Thus in Figure 2.a, module e_1 is adjacent to modules e_5, e_6, e_7, e_8, and e_9 and is rectilinearly adjacent to modules e_5, e_6, and e_7.

3.2. Scoring Function σ

To aid in the comparison of our algorithm with other algorithms, all instances were evaluated with respect to a variant of the scoring function used by Kirkpatrick, Gelatt and Vecchi's simulated annealing placement algorithm [KIRK83]. Informally, the scoring function was an aggregate of the half perimeter of the bounding rectangle for each net and of the excess horizontal and vertical channel usage. A formal specification follows. The routing subregion between two consecutive rows is a *horizontal channel* and the routing subregion between two consecutive vertical rows is a *vertical channel*. Let $\square_i$ be the size of the perimeter for the bounding rectangle of net i, $1 \le i \le n$, where the length of a side is the number of channels it crosses and n is the number of nets. Let h_i be the number of nets whose bounding rectangle crosses horizontal channel i, $1 \le i \le r-1$. Let v_i be the number of nets whose bounding rectangle crosses vertical channel i, $1 \le i \le c-1$. Let $\bar{h}$ and $\bar{v}$ be respectively the mean values of the h_i's and the v_i's. Let s_h and s_v be measures respectively of one standard deviation of horizontal and vertical channel usage i.e.

$$s_h = \left[\sum_{i=1}^{r-1} \frac{(h_i-\bar{h})^2}{r-2} \right]^{\frac{1}{2}}$$

and

$$s_v = \left[\sum_{i=1}^{c-1} \frac{(v_i-\bar{v})^2}{c-2} \right]^{\frac{1}{2}}.$$

Let $\hat{h}_i$ and $\hat{v}_j$ be measures respectively of excessive usage (if any) in horizontal channel i and vertical channel j i.e.

$$\hat{h}_i = \begin{cases} h_i-\bar{h}-s_h & \text{if } \bar{h}+s_h < h_i \\ 0 & \text{otherwise} \end{cases}$$

and

$$\hat{v}_j = \begin{cases} v_j-\bar{v}-s_v & \text{if } \bar{v}+s_v < v_j \\ 0 & \text{otherwise} \end{cases}.$$

The scoring function σ returns

$$\tfrac{1}{2}\sum_{i=1}^{n}\square_i + \sum_{i=1}^{r-1}\hat{h}_i^2 + \sum_{j=1}^{c-1}\hat{v}_j^2. \qquad (3.1)$$

Such a function encourages short interconnections with even horizontal and vertical channel capacities. For standard cell placement, we recommend that the score be reduced when adjacent cells in a row belong to the same net. This reduction is applied since it is often possible to connect such cells without going into a channel.

3.3. Population Constructor Ξ

Three population constructors Ξ_1, Ξ_2 and Ξ_3 were initially considered. Constructor Ξ_1 randomly places modules; constructors Ξ_2 and Ξ_3 attempt to group modules in the same net to improve the scores of the initial population. As all three constructors had a randomizing component they could be repeatedly applied to produce an initial population of desired size.

Constructors Ξ_2 and Ξ_3 choose the order that the modules are assigned in the same manner. Their processes resemble the repeated folding of a 1-dimensional placement. Their constructive process starts by selecting a net at random and choosing its modules in net list order. Let i be the number of modules assigned so far. Next, an unselected net containing the *i-th* placed module is chosen and its unplaced modules are assigned locations. If no such net exists, then an unselected net containing the $i-1$-*st* placed module is sought, and so on. If there are no unselected nets containing modules already placed, an unselected net is chosen at random and the process continues until all modules have been placed.

The difference between Ξ_2 and Ξ_3 is the order the slots are assigned. Constructor Ξ_2 places modules in boustrophedon fashion (Figure 3.a) and constructor Ξ_3 places modules in row-major fashion (Figure 3.b).

One difference between our genetic algorithm and the algorithm of Figure 1 is that our algorithm does not iterate for some fixed number of generations. Instead, our algorithm terminates once the population is homogeneous with respect to the best solution score (i.e. if the best score has not improved in k generations, the algorithm terminates). We found that the choice of constructor affected both the quality of the solution and the number of generations needed to generate a homogeneous population. Populations constructed by Ξ_1 tended to have a slower rate of convergence towards homogeneity. Populations constructed by Ξ_2 had better average initial scores, but tended to converge almost immediately to inferior local minima. We believe this occurred because the solutions produced by Ξ_2 were good, but often far

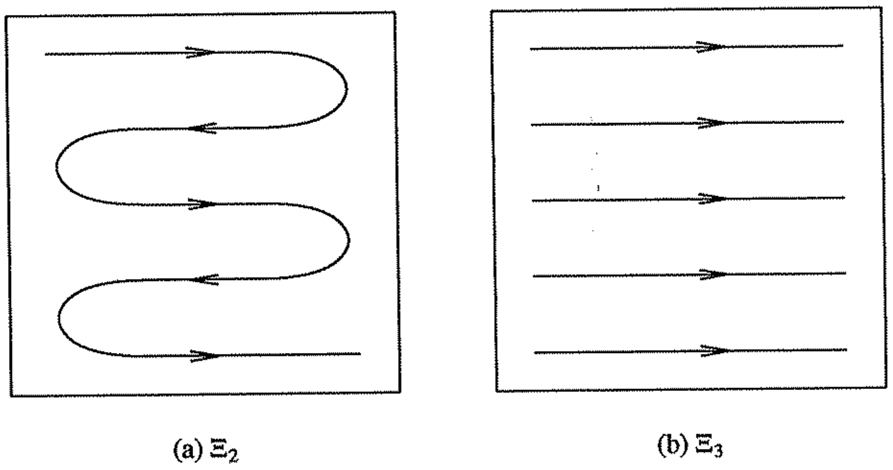

(a) Ξ_2 (b) Ξ_3

Figure 3 — Order of Slot Assignment

from optimal and necessitated a great deal of hill-climbing to reach an optimum. There was so much score degradation as a result of the hill-climbing, that the solutions tended to die-off before they could improve. This led us to try Ξ_3, which attempts to group modules in the same net, but not necessarily as closely as Ξ_2 does. As a further conservative measure, we did not strictly use Ξ_3 to construct the initial population. Our experiments indicated that a mixed initial population—75% of the initial placements constructed by Ξ_3 and the remaining 25% constructed by Ξ_1—resulted in a good initial mean population score while maintaining a satisfactory amount of diversity. Table 1 illustrates the results of several combinations of Ξ_1, Ξ_2, and Ξ_3 on a problem instance with 81 modules to be placed on a chip with 9 rows and 9 columns.

3.4. Crossover Operator ψ

Our initial investigation of crossover operators considered two types. One combined the two parent strings to form the offspring string by using information passing [GOLD85]; the other created the offspring by using one parent to define a rearrangement of the other parent [GREF85a]. We concluded that operators of the first type were more amenable to directed-evolution principles. Therefore, the majority of our time was spent analyzing such crossover operators. From this analysis, two crossover operators, ψ_1 and ψ_2, were derived. Both ψ_1 and ψ_2 perform a "cut-paste-and-patch" with the two parents. One parent is selected randomly to be the basis of the offspring. It is called the *target* parent. The other parent is called the

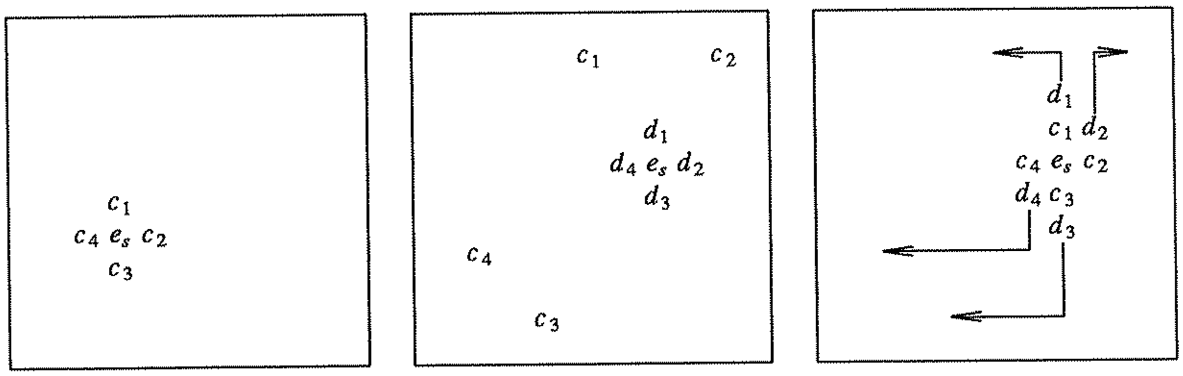
Comparison of Population Constructors				
Score	Initial Population Constructor			
	100% Ξ_1	100% Ξ_2	25% Ξ_1 75% Ξ_2	25% Ξ_1 75% Ξ_3
Initial (average)	517	329	379	423
Final (best)	120	186	169	120
Iterations	55,000	35,000	40,000	20,000

Table 1

passing parent. A portion of its alleles are "cut" from it and "pasted" into a copy of the target parent. The copy is then "patched-up" to make it a legal solution.

Let α_x and α_y be respectively the passing and target parents. Operator ψ_1 operates in the following manner. An identical copy α_o of α_y is created. A module e_s is randomly selected from M. Its location is determined in α_x and α_y. Let $c_1, \ldots, c_4$ and $d_1, \ldots, d_4$ be the adjacent modules above, right, below, and left of e_s in respectively α_x and α_y. The goal of ψ_1 is to reconfigure offspring α_o such that modules $c_1, \ldots, c_4$ occupy respectively the starting slots of $d_1, \ldots, d_4$ while minimally perturbing the other modules. The reconfiguration is done in the order c_1, c_3, c_2, and c_4 and uses a "sliding" process to make room for $c_1, \ldots, c_4$. A graphical representation of ψ_1's operation is given in Figure 4. The arrows in Figure 4.c do not point to the d_i's final locations. They instead indicate the other modules that are affected by the crossover.

The sliding process begins by determining a sequence or a *chain* of modules from d_1 to c_1. Figure 5.a shows some of the forms the chains would take if c_1 is either to the left or below e_s. The remaining cases are analogous. If c_1 is located above e_s's row, then the chain extends vertically from d_1 to c_1's row, and then horizontally to c_1. If c_1 is either on or below e_s's row and to its left (right), then the chain extends left (right) one column from d_1, then down to c_1's row, and then horizontally to c_1. If instead c_1 lies below e_s in e_s's column, then the path of chain extends randomly from d_1 either right or left one column, then precedes vertically to c_1's row, and then horizontally one column to c_1's location. Once the

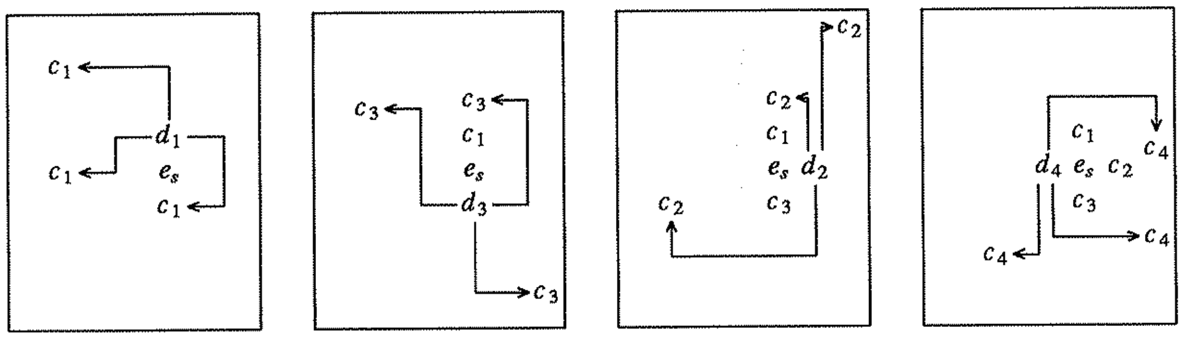


(a) Passing Parent (b) Target Parent (c) Offspring

Figure 4 — Crossover Operator ψ_1

chain is established, its modules are iteratively shifted along its path away from d_1 toward c_1. Finally, c_1 is moved into the unoccupied location above e_s. The module shifting terminates once either a module is assigned to an unoccupied location or a module is assigned c_1's location.

Once c_1 is moved into its new location, c_3 is moved directly below e_s. The chain for this process is created analogously to the chain for c_1. Figure 5.b shows several of the possible chain-forms from d_3 to c_3.



(a) (b) (c) (d)

Figure 5 — Some Pathological ψ_1 Sliding Cases and Their Resolution

Moving c_2 and c_4 is complicated by the need to avoid perturbing e_s, c_1, and c_3 as they are already in their correct locations. Figure 5.c shows several of the possible chain-forms from d_2 to c_2. If c_2 is either to the right of e_s or if it lies above c_1's row or below c_3's row, then the chain extends vertically from d_2 to c_2's row, and then horizontally to c_2's location. If instead c_2 is to the left of e_s and lies on c_1, c_3 or e_s's row, then a horseshoe-like chain (i.e. a vertical-horizontal-vertical move) is created that passes over c_1 or under c_3. If c_2 lies on either e_s or c_1's row, the horseshoe chain then passes above c_1, unless c_1 lies in the top row where the chain instead must pass below c_3. Similarly, if c_2 lies on c_3's row, the horseshoe chain then passes below c_3, unless c_3 lies in the bottom row where the chain instead must pass above c_1.

Finally, c_4 is moved into d_4's location by creating a chain similar to c_2's. Figure 5.d shows several of the possible chain-forms from d_4 to c_4.

Unlike ψ_1's operation, ψ_2 does not pass a constant number of alleles from the passing parent α_x to the offspring α_o. Instead a $k \times k$ square section C of α_x is copied into the offspring, where variate k has a truncated normal distribution with mean 3 and variance 1. Let D be the square section of modules in α_o that occupies the location corresponding to C's location (α_o is again previously initialized to be a copy of target parent α_y). Let $D - C$ be the modules in square D but not in square C. Let $C - D$ be the modules in square C but not in square D. Each module in $D - C$ is moved to a slot currently occupied by a module in $C - D$. Once all such modules are moved, square C is copied into square D.

Operator ψ_2's operation is shown graphically in Figures 6.a-6.d. In Figure 6.a, a 3×3 square C has been selected for passing. In Figure 6.b, both the corresponding square D and the modules in $C - D$ have been determined. Suppose $D - C = \{d_1, d_2, d_6, d_7, d_8, d_9\}$. Figure 6.c shows the repositioning of the modules in $D - C$. In Figure 6.d, the offspring is completed with the copying of C into D.

Although, both operators ψ_1 and ψ_2 performed well during experimental analysis, ψ_2 was consistently better. Therefore, ψ_2 was incorporated into our final algorithm. Table 2 summarizes several runs of our algorithm using ψ_1 and ψ_2.

3.5. Choice Function ϕ

Four choice functions, $\phi_1, \ldots, \phi_4$, were considered for selecting parent strings from the population. Each function was tested using ψ_2 as the crossover operator. Function ϕ_1 chooses a random string and the best

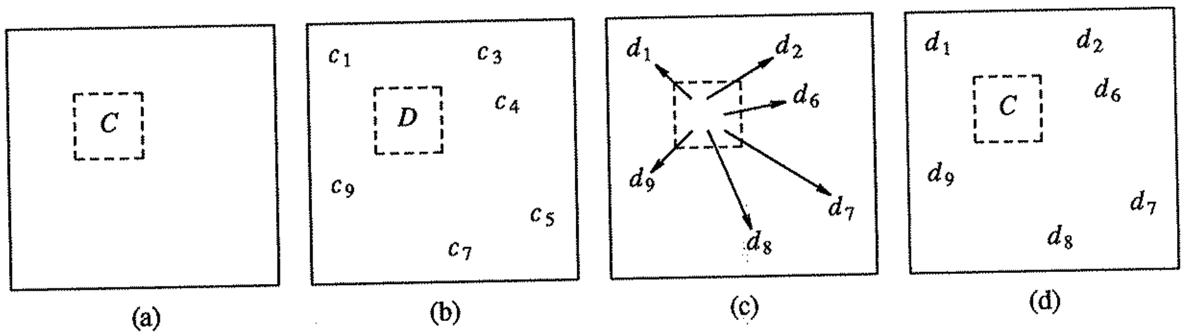


Figure 6 — Crossover Operator ψ_2

Contrasting Crossover Operator's Performance	
Average Solution Score over 25 Runs	
Using ψ_1	Using ψ_2
303	258

Table 2

scoring string as parents. The result was a quick marked loss of diversity within the population; after fewer than 500 iterations the great majority of the strings closely resembled the best string in the initial population. Final scores were poor as a result of convergence to an inferior local minimum.

Function ϕ_2 randomly chooses both parents from the population. The effect was just the opposite of ϕ_1; there was little or no convergence to a good solution, and after tens of thousands of generations the average score of the population was often no better than that of the initial population.

Functions ϕ_3 and ϕ_4 are syntheses of ϕ_1 and ϕ_2 that favor the better scoring strings as parents. Let S be a set of strings from Π. Let $s \in S$. The *weight* of s with respect to S is

$$w(s,S) = \frac{\max\{\,\sigma(t) \mid t \in S\,\}}{\sigma(s)}.$$

Let $\overline{w}(S)$ be the average weight of the strings in S. Let

$$\hat{w}(s,S) = \begin{cases} w(s,S) & \text{if } w(s,S) \le \overline{w}(S) \\ 0 & \text{otherwise} \end{cases}.$$

Let $\{\alpha_1, \ldots, \alpha_p\}$ be the set of strings in P. String α_j, $1 \le j \le p$, is chosen by ϕ_3 for mating with probability

$$\frac{w(\alpha_j,P)}{\sum_{i=1}^{p} w(\alpha_i,P)}$$

and is chosen by ϕ_4 for mating with probability

$$\frac{\hat{w}(\alpha_j,P)}{\sum_{i=1}^{p} \hat{w}(\alpha_i,P)}.$$

Thus, although both ϕ_3 and ϕ_4 probabilistically favor the better scoring strings, ϕ_4 further requires that the parent's score be better than average. Both ϕ_3 and ϕ_4 performed well. However, we found that ϕ_4 was marginally better. Therefore, ϕ_4 was incorporated into our final algorithm.

3.6. Selector ρ

Three selectors, ρ_1, ρ_2, and ρ_3 were considered for determining which strings to use as the basis of the next generation. Let $Q = P \cup Offspring$. Selector ρ_1 chooses the best scoring string from Q and randomly chooses $p-1$ other strings from Q. Its results were so discouraging that we omitted the implementation of a selector that chooses the p best scores, since we believe the results would be the same or worse than those for ρ_1. Selector ρ_2 randomly chooses p strings from Q to form the next generation of the population. As a result, the best scoring string is sometimes not selected to survive to the next generation and its loss means the sequence of the best score in progressive generations is not likely to be monotonically decreasing. On the other hand, random selection helps prevent early convergence to inferior solutions that closely resemble the early best scoring strings. Let $Q = \{\alpha_1, \ldots, \alpha_q\}$. The probability that ρ_3 selects α_j, $1 \le j \le q$, is

$$\frac{w(\alpha_j,Q)}{\sum_{i=1}^{q} w(\alpha_i,Q)}.$$

Thus, each string in Q has a non-zero probability of surviving with ρ_3 as the selector. In our final algorithms, we used both ρ_2 and ρ_3 depending on the characteristics of the data instances.

3.7. Mutation Function μ

Two mutation operators μ_1 and μ_2 were considered. Operator μ_1 mutates a solution by performing a series of random *interchanges*, where an interchange takes a pair of modules and swaps their locations. Its random nature allows for a broader exploration of the state-space. However, we found that this operator typically increased a string's score due its disruptive effect on placements. Figures 7.a and 7.b show a placement before and after modules e_4 and e_8 have swapped locations.

Operator μ_2 uses a directed-evolution transformation that attempts to reduce the bounding rectangle of a randomly selected net Z while perturbing the other nets as little as possible. It does this by selecting a module e_s at random from Z and then relocating the module e_t in Z that is farthest from it to a closer location, where the distance between modules is the rectilinear distance.

The *preferred* location of e_t is a location rectilinearly adjacent to e_s that lies on a minimal distance path from e_s to e_t. Let (x_s, y_s) and (x_t, y_t) be respectively the locations of e_s and e_t. Let $\Delta x = |x_s - x_t|$ and $\Delta y = |y_s - y_t|$. If $\Delta x = 0$ or $\Delta y = 0$ then there is only one such location. If instead $\Delta x \neq 0$ and $\Delta y \neq 0$ then there can be two modules e_c and e_d that are rectilinearly adjacent to e_s that lie on minimal distance paths from e_s to e_t. Let e_c be the one with y-coordinate y_s and e_d be the one with x-coordinate x_s. If $|\Delta x| \leq |\Delta y|$ then the preferred location is e_d's, otherwise the preferred location is e_c's. For example, in Figure 8.a modules a_1 and a_3 correspond respectively to e_c and e_d. If we assume for this figure that $0 < |\Delta y| < |\Delta x|$ then the preferred location for e_t is module a_3's.

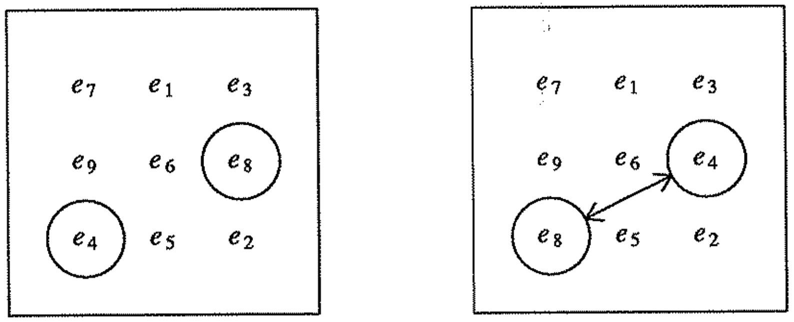


Figure 7 — A Pairwise Swap Mutation Move

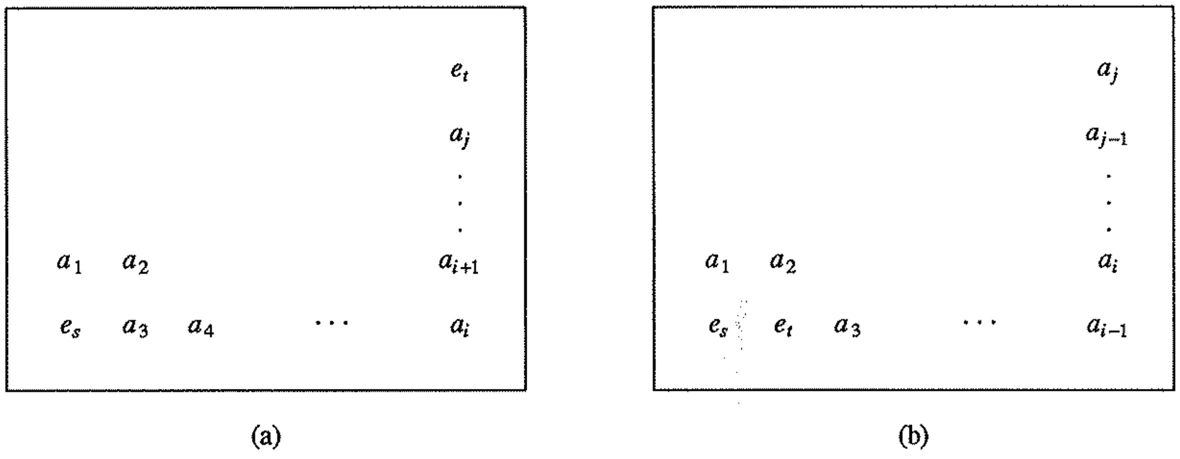


Figure 8 — A Directed-Evolution Mutation Move

As discussed, the goal of μ_2 is to reduce the size of the bounding rectangle of the net on that it operates. This goal can be further advanced by determining if the module currently assigned to the preferred location also belongs to net Z. If this does occur and e_c (e_d) is the preferred location then the new preferred location is e_d (e_c). In Figure 8.a, if a_3 belongs to net Z then the new preferred location of e_t contains a_1. If this location also contains a module in net Z, then the preferred location is finally changed to the location (other than e_s's location) that is adjacent to both e_c and e_d. In Figure 8.a, if a_1 and a_3 belong to net Z then the preferred location contains module a_2.

Once the preferred location has been determined, a sliding process similar to ψ_1's operation is used to relocate e_t next to e_s. If the preferred location for e_t lies in the same column as e_s, the chain extends first vertically and then horizontally, otherwise the chain extends first horizontally and then vertically. Figure 8.b shows the reconfiguration of the placement after both the chain has been moved and e_t has been assigned to a_3's old location.

In practice, we had mutation operator μ_2 move two modules in a net, rather than a single module. Once e_t has been moved closer to e_s, the new farthest module from e_s is determined and the process is repeated. Therefore, in nets of two or three modules, μ_2 reduces the size of the bounding rectangle of the selected net to the minimum possible.

Table 3 illustrates a representative change in score resultant from μ_1 and μ_2 on a given placement with 81 modules arranged in 9 rows and 9 columns. The two mutators were both applied sequentially 100 times on the placement. 29 of μ_1's applications improved the placement while 64 degraded it. On average, a μ_1 mutation degraded the score by 4.2 units. This contrasts with μ_2 that performed one more improvement than degradation and where the average change was an improvement of 0.2 units. Table 4 summarizes several runs using μ_1 and μ_2. The final solutions found using μ_2 were on average 18% better. As a result of our preliminary experiments, μ_2 was included in our final algorithm.

In an attempt to determine the relative contributions of mutator μ_2 and crossover operator ψ_2 to the final solution, we ran several experiments where both operators were used and several experiments where only one of these two operators were used. The results were interesting. Although the genetic algorithm consistently determined optimal solutions when using both operators, when it used only one of two there was significant solution degradation. The amount of degradation was about 100% for both cases. Thus, we

Changes in Score after 100 Applications of μ_1 and μ_2

Operator	Improvements		Degradations		Overall Average Change
	Number	Average Change	Number	Average Change	
μ_1	29	4.3	64	-8.5	-4.2
μ_2	44	7.8	43	-7.5	0.2

Table 3

Contrasting Mutation Operator's Performance

Average Solution Score over 25 Runs	
Using μ_1	Using μ_3
300	245

Table 4

conclude there is wholistic effect when the two operators are used together.

3.8. ALGORITHM GENIE

As a result of our experimental analysis, algorithm *Genie* was derived. To speed-up the mutation process *Genie* does not individually consider whether each string should be mutated. Instead, the algorithm draws a variate k that indicates how many strings should be mutated. Variate k is normally distributed with mean $K_\mu \cdot p \cdot n$ and variance 1. After determining k, *Genie* randomly chooses the k strings to be mutated. As repetition is permitted, a string may be mutated more than once. Although the best solutions were determined with K_μ and K_ψ (the proportion of the population involved in mating) set respectively to 0.009 and 0.25, we suggest several runs be made with varying values for the two parameters. Two versions of *Genie* were implemented—$Genie_1$ and $Genie_2$. The two versions differ only in their selectors. $Genie_1$ uses selector ρ_2 and $Genie_2$ uses selector ρ_3. Several population sizes were also tested. The smallest population size considered was 10 and the largest size considered was 1000. *Genie* performed best with the population size p set to 50. This result agrees with Grefenstette's investigation of control parameters for genetic algorithms [GREF86]. Grefenstette found that larger population sizes (greater than 100) adversely effected the rate of population convergence and that smaller populations sizes (less than 30) were more volatile with respect to solution quality. A high-level description of $Genie_1$ is given in Figure 9.

```
 1. $P \leftarrow \Xi_1(M, N, 0.25p) \cup \Xi_3(M, N, 0.75p)$
 2. while best score has not improved in 10,000 generations do
 3.      $Offspring \leftarrow \varnothing$
 4.      for $k \leftarrow 1$ to $p \cdot K_\psi$ do
 5.           $(x, y) \leftarrow \phi_4(P)$
 6.           $Offspring \leftarrow Offspring \cup \{\psi_2(x, y)\}$
 7.      end for
 8.      $P \leftarrow \rho_2(P, Offspring)$
 9.      $k \leftarrow random(normal, K_\mu p\, n, 1)$
10.      for $i \leftarrow 1$ to $k$ do
11.           $x \leftarrow random(uniform, P)$
12.           $x \leftarrow \mu_2(x)$
13.      end for
14. end while
```

Figure 9 — Algorithm $Genie_1$

4. PERFORMANCE

Five circuit instances, $D_1, \ldots, D_5$, were used to analyze the performance of *Genie*. Solution quality was our primary concern; run-time was of secondary interest. As a comparison point, we also implemented the original simulated annealing placement algorithm (*SP*) of Kirkpatrick, Gelatt, and Vecchi [KIRK83]. We chose to compare *Genie* with *SP* for three reasons: 1) *SP* is easy to implement; 2) *SP* produces very good solutions; and 3) it allows us to contrast an initial attempt at genetic algorithm placement with the initial version of another methodology. However, we note that in choosing *SP*, we selected an algorithm that is inferior to newer simulated annealing placement algorithms that have more advanced annealing schedules and solution perturbation techniques. For example, Sechen and Sangiovanni-Vincentelli's TimberWolf [SECH84] offers comparable or improved solution quality and it is estimated to be an order of magnitude faster than previous simulated annealing attempts. Thus, our run-time comparisons with simulated annealing are qualified.

Some characteristics of the five basic data instances are provided in Table 5. Another notable characteristic of a data instance is the amount of *structure* it contains. When placed in an optimal or nearly optimal fashion, structured data results in placements containing tightly coupled repeating patterns of modules. Non-structured data results in placements with few or no patterns. The five test instances specified in Table 5 are ranked in decreasing order of structure so that D_1 has the most structure and D_5 has the least. A small example of an optimal placement of a very-structured data instance is given in Figure 10. In the figure the modules are denoted by circles and the interconnections by lines. A solution is

Characteristics of the Data Instances

Instance	Number			
	Modules	Nets	Rows	Columns
D_1	50	32	5	10
D_2	36	29	6	6
D_3	81	57	9	9
D_4	84	153	12	7
D_5	48	53	6	8

Table 5

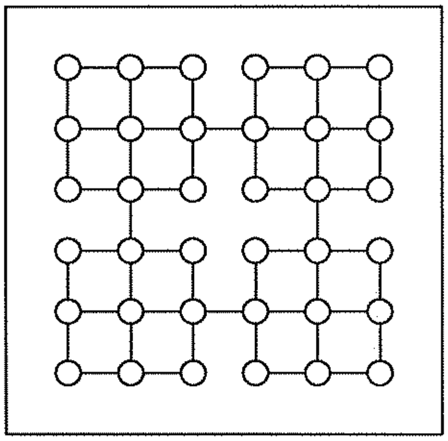


Figure 10 — Optimal Placement of a Structured Data Instance

known to be optimal if each net is laid out in a minimal bounding rectangle and the second and third terms of the scoring function (Equation 3.1), which measure the unevenness of horizontal and vertical channel densities, are zero. We observed that the best placements of structured data instances are usually formed in two stages. First the objects are moved into the "correct" area on the board and then the local areas are simultaneously fine-tuned.

Our preliminary experiments indicated that the degree of data instance structure determines whether *Genie*$_1$ or *Genie*$_2$ will find the better solution. Highly non-structured data benefits from the directed search of *Genie*$_2$ that encourages the survival of better scoring strings to the next generation. But this favoring of better strings leads to premature convergence to inferior local minima for highly structured data. In nearly every case of premature convergence, the modules were fine-tuned correctly relative to their adjacent modules, but tended not to be located in the proper area of the board. Thus structured data benefits from the less directed search of *Genie*$_1$ that randomly selects strings to survive to the next generation.

The above observation led us to make an additional change to the *Genie* algorithms, specifically to the mutation function μ_2. If the modules in the net chosen for mutation are already adjacent, the net may be left unchanged by μ_2. Since the purpose of the mutation operator is to make a change in the string, we modified the *Genie* algorithms to mutate a string that would have been unperturbed by μ_2 by swapping the positions of two modules in the chosen net. Therefore, we have a third version of *Genie*, *Genie*$_1$*, that is *Genie*$_1$ with the modified mutation move. A fourth version, *Genie*$_2$*, with probabilistic selector ρ_3 and the modified mutation operator, was also considered but it consistently produced poor results.

A summary of the relative performance of *Genie* and *SP* for the data instances $D_1, \ldots, D_5$ is given in Table 6. For the genetic approach, we used algorithm *Genie*$_1$* for instances D_1 D_2 and D_3, and *Genie*$_3$ for instances D_4 and D_5. Each appropriate *Genie* and *SP* algorithm was run several times per instance with only the best scoring solutions reported. The table indicates that both algorithms performed well. For the three instances D_1, D_2, and D_3 with known optimal solutions, *Genie* found all three and *SP* found two of the three. For instance D_4, *Genie*'s solution was better by 10 units then *SP*'s solution. For the final instance D_5, *Genie* and *SP* produced virtually identical solutions.

As the algorithms were run on several different machines, we supply in Table 6 the approximate number of placements examined by each algorithm rather than reporting run-time directly. Although the difference in the number of placements is consistently in *Genie*'s favor, the difference in run-times is not. This is due to the additional overhead that *Genie* incurs in determining which placements to select and manipulate in its various operations. For example, in one problem instance *Genie* required approximately 50 seconds on a Sun-2 workstation to iterate for enough generations to construct 1000 placements while *SP* required less than 30 seconds to generate and evaluate 1000 placements. Hence, we believe that *SP* was

Solution and Run-Time Comparison					
Data	Optimal	*Genie*		*SP*	
Instance	Score	Best Score	Placements Examined	Best Score	Placements Examined
D_1	60	60	395,000	60	750,000
D_2	46	46	375,000	46	500,000
D_3	120	120	1,150,000	128	1,500,000
D_4	?	226	3,000,000	236	6,000,000
D_5	?	156	540,000	157	1,900,000

Table 6

more effective for instance D_3, *Genie* was more effective for instances D_4 and D_5, and that the algorithms were equally effective on instances D_1 and D_2. It is an open question whether refined versions of genetic algorithms will be able to compete with the more sophisticated simulated annealing algorithms such as TimberWolf with annealing schedules that are much less conservative then *SP*'s schedule.

Since we desired to extensively explore and compare genetic algorithm alternatives among themselves and simulated annealing, and since we had limited computational resources, small instances were our primary test cases. However, we recognize that if *Genie* is to be used, it must perform well on much larger instances. Part of our current research effort is directed at making *Genie* an effective tool for such problem sizes. We are currently considering several medium-sized instances that range in module count from 500 to over a 1000. We expect to use these results to help guide the development of a version of *Genie* that will be competitive on problem sizes on the order of 10^4. At this point, our experience on these medium-sized problems has been limited to instances with known optimal or near-optimal solutions. *Genie*'s preliminary performance is encouraging, but less effective than its performance on smaller problem instances. The increase in run-time was at most a linear factor in the problem size. For example, while 3 million solutions were examined by *Genie* to solve D_3 with its 84 modules, when *Genie* examined an instance with 1020 modules, it considered (only!) 19 million solutions. For some of the instances *Genie* was able to find solutions with scores only 13% greater than optimal. However, *Genie* also produced scores 50% greater than the optimal solution. This is in sharp contrast to our experience with smaller instances where the difference between runs was on the order of 2½ percent. Therefore, we are directing our efforts to reduce this volatility. One probable action is further evaluation of crossover techniques. The current crossover operator may not be sufficiently rigorous when trying new subarrangements of the modules. We also intend to experiment more with larger populations. This would allow more and diverse placement "ideas" to be simultaneously considered by the genetic algorithm.

5. SUMMARY

Since placement is such an important component of circuit layout, it has been examined extensively and many diverse approaches have been successfully proposed. These approaches can for the most part be loosely classified as either constructive or iterative improvement algorithms. Although genetic algorithms resemble previous iterative improvement techniques, there is one fundamental difference—they act on a set of solutions rather than a single solution. This feature along with their usage of crossover and mutation operators gives genetic algorithms the unique ability to explore and combine simultaneously diverse placement ideas in a variety of situations. This ability when combined with the probabilistic nature of its remaining operators and functions allows a speedy directed search of the state space towards those portions with desirable characteristics. Since we find this method of operation to be intuitively appealing, we initiated a research effort that resulted in *Genie*.

During our experimental analysis of *Genie*, we examined its performance on several problem instances. Its performance on the smaller instances were observed to be uniformly good (e.g. for three of the instances it found optimal solutions). For these same instances, we also compared *Genie*'s performance to the original simulated annealing algorithm *SP* [KIRK83]. We found *Genie*'s performance to be competitive with *SP*'s performance (e.g., its solutions in all instances were as good or better than *SP*'s solutions). We have also examined—though in less detail—*Genie*'s performance on some larger instances. Its solutions here were not as good as its solutions for the smaller instances. Although it found solutions with scores within 13% of the optimal score, solution quality was far more volatile (e.g. it also returned solutions with scores 50% higher than the optimal score). Thus, although we believe our experiments indicate that *Genie* is a promising placement technique, they also indicate that additional research is in order. This research must answer such questions as whether *Genie* can be refined to be competitive with improved simulated annealing algorithms and whether it can be tuned to produce consistently good solutions for larger problem instances.

Our research effort with *Genie* is still ongoing. Besides attempting to answer the questions posed above, we are also developing a distributed formulation of genetic algorithms [COHO87]. The same non-conventional properties that appealed to us as a possible placement technique lead us to believe it is a good candidate for parallelization.

6. ACKNOWLEDGEMENTS

The comments and suggestions of Alberto Sangiovanni-Vincentelli and the referees were greatly appreciated. Their remarks substantially aided the quality of discussion in several sections of this paper.

The authors' work has been supported in part by National Science Foundation grant DMC-8505354, Virginia Center for Innovative Technology grant INF-86-001, and a General Electric Microelectronics Center grant. Their support is also greatly appreciated.

7. REFERENCES

[BREU77] M. A. Breuer, Min-Cut Placement, *Design Automation & Fault-Tolerant Computing 1*, 4 (October 1977), 343-362.

[CHEN84] C. K. Cheng and E. S. Kuh, Module Placement Based on Resistive Network Optimization, *IEEE Transactions on Computer-Aided Design CAD-3*, 3 (July 1984), 218-225.

[COHO87] J. P. Cohoon, S. U. Hegde, W. N. Martin and D. S. Richards, Punctuated Equilibria: A Parallel Genetic Algorithm, *Proceedings of the Second International Conference on Genetic Algorithms and Their Applications*, Boston, MA, 1987, 148-154.

[DUNL85] A. E. Dunlop and B. W. Kernighan, A Procedure for Placement of Standard-Cell VLSI Circuits, *IEEE Transactions on Computer-Aided Design CAD-4*, 1 (January 1985), 92-98.

[GOLD85] D. E. Goldberg and R. Lingle,Jr., Alleles, Loci, and the Traveling Salesperson Problem, *Proceedings of an International Conference on Genetic Algorithms and Their Applications*, Pittsburgh, PA, 1985, 154-159.

[GREF85a] J. J. Grefenstette, R. Gopal, B. J. Rosmaita and D. VanGucht, Genetic Algorithms for the Traveling Salesperson Problem, *Proceedings of an International Conference on Genetic Algorithms and Their Applications*, Pittsburgh, PA, 1985, 160-168.

[GREF85b] J. J. Grefenstette, ed., *Proceedings of an International Conference on Genetic Algorithms and Their Applications*, Pittsburgh, PA, 1985.

[GREF86] J. J. Grefenstette, Optimization of Control Parameters for Genetic Algorithms, *IEEE Transactions on Systems, Man and Cybernetics SMC-16*, (1986), 122-128.

[HANA72] M. Hanan and J. M. Kurtzberg, A Review of the Placement and Quadratic Assignment Problems, *SIAM Review 14*, 2 (April 1972), 324-342.

[HOLL75] J. H. Holland, *Adaptation in Natural and Artificial Systems*, University of Michigan Press, Ann Arbor, MI, 1975.

[KIRK83] S. Kirkpatrick, C. D. Gelatt and M. P. Vecchi, Optimization by Simulated Annealing, *Science 220*, 4598 (May 13, 1983), 671-680.

[QUIN75] N. R. Quinn, The Placement Problem as Viewed from the Physics of Classical Mechanics, *12th Design Automation Conference Proceedings*, Boston, MA, 1975, 173-178.

[SECH84] C. Sechen and A. Sangiovanni-Vincentelli, The TimberWolf Placement and Routing Package, *Proceedings of the Custom Integrated Circuit Conference*, , 1984.