Genetic Placement

JAMES P. COHOON, MEMBER, IEEE, AND WILLIAM D. PARIS

His genie led him into the pleasant paths.

—Anthony Wood, 1662

***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 variety of small test instances. Its solutions were observed to be quite good and in several cases optimal.**

***Keywords*—Placement, genetic algorithms, VLSI, physical design.**

I. 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*, $M = \{e_1, \cdots, e_m\}$, and a set of n signals or *nets*, $N = \{s_1, \cdots, s_n\}$, 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*, $L = \{c_1, \cdots, c_l\}$, 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.

Manuscript received December 22, 1986; revised May 28, 1987. This work was supported in part by the National Science Foundation under Grant DMC-8505354, the Virginia Center for Innovative Technology under Grant INF-86-001, and a General Electric Microelectronics Center Grant.

J. P. Cohoon is with the Department of Computer Science, University of Virginia, Charlottesville, VA 22903.

W. D. Paris was with the Department of Computer Science, University of Virginia, Charlottesville. He is now with Syntek Systems Incorporated, Bethesda, MD.

IEEE Log Number 8716061.

Because of its importance, the placement problem has received considerable attention and many diverse solution strategies have been proposed: partitioning [1], [2], quadratic assignment [3], force-directed [4], resistive network [5], and simulated annealing [6].

The contribution presented here is a new placement algorithm, *Genie*, which is based on the genetic algorithm approach [7]. 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 [8]. Based on our preliminary experiments, this new use of genetic algorithms appears to be a promising method of solving placement problems.

II. 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 σ: $\Pi \rightarrow R^+$. 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, ϕ: $2^\Pi \rightarrow \Pi \times \Pi$, 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, ψ: $\Pi \times \Pi$, is known as *crossover*. After the crossovers are performed, a selector ρ: $2^\Pi \times 2^\Pi \rightarrow 2^\Pi$ is applied to the previous generation and the offspring to determine which

```
1.  P ← initial population constructed with function Ξ
2.  p ← |P|
3.  for i ← 1 to NUMBER_GENERATIONS do
4.      Offspring ← ∅
5.      for k ← 1 to p · K_ψ do
6.          (x, y) ← φ(P)
7.          Offspring ← Offspring ∪ {ψ(x, y)}
8.      end for
9.      P ← ρ(P, Offspring)
10.     for each string x∈P do
11.         with probability K_μ mutate x with μ
12.     end for
13. end for
14. return highest scoring string in P
```

Fig. 1. Genetic algorithm paradigm.

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*, μ: $\Pi \rightarrow \Pi$. 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 Fig. 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.

III. 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.

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

The encoding ξ_1 in Fig. 2(b) divides each allele into two fields, I_j and M_j, $1 \leq j \leq l$. Field I_j specifies a location on the chip and field M_j specifies the index of the module in I_jth location. Thus, the third allele in Fig. 2(b) specifies that module e_6 is found in the fifth location. The encoding ξ_2 in Fig. 2(c) uses a single field R_j for each allele j, $1 \leq j \leq l$. R_j is used to calculate the module located in the jth location on the chip. Besides the obvious function that directly used R_j as the index of the module in the jth location, we considered a variant of the ordinal function of Grefenstette, Gopal, Rosmaita, and VanGucht [9]. It works as follows: the module with the R_1th smallest index with respect to all modules is assigned the first location; the module with the R_2th 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 Fig. 2(c) contains a 4, module e_4 with the 4th 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 8th 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 6th 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 jth allele specified the index of the module in the jth location.

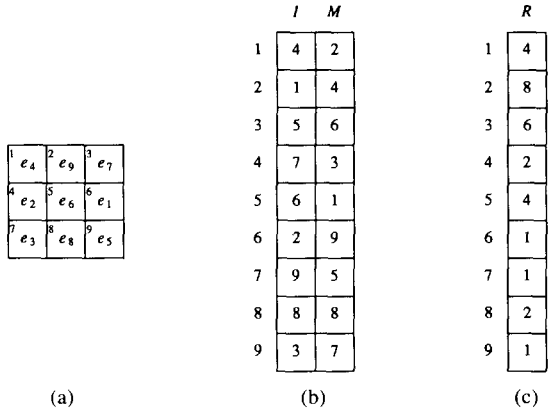

Fig. 2. Some string encoding. (a) Placement. (b) ξ_1. (c) ξ_2.

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 Fig. 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.

B. 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 [6]. 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, $1 \leq i \leq n$, 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, $1 \leq i \leq r-1$. Let v_i be the number of nets whose bounding rectangle crosses vertical channel i, $1 \leq i \leq c-1$. 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.,

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

and

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

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

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

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.

C. 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 one-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 ith 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.

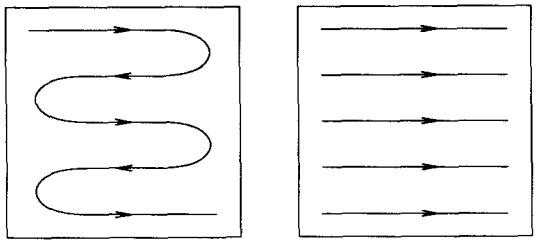

Fig. 3. Order of slot assignment. (a) Ξ_2. (b) Ξ_3.

The difference between Ξ_2 and Ξ_3 is the order the slots are assigned. Constructor Ξt_2 places modules in boustrophedon fashion (Fig. 3(a)) and constructor Ξ_3 places modules in row-major fashion (Fig. 3(b)).

One difference between our genetic algorithm and the algorithm of Fig. 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 by Ξ_2 were good, but often were far 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 lead 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 inital population. Our experiments indicated that a mixed initial population—75 percent of the initial placements constructed by Ξ_3 and the remaining 25 percent constructed by Ξ_1—resulted in a good initial mean population score while maintaining a satisfactory amount of diversity. Table I 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 nine rows and nine columns.

TABLE I
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

D. 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 [10]; the other created the offspring by using one parent to define a rearrangement of the other parent [9]. 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, 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 *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 $c_1, \cdots, c_4$ and $d_1, \cdots, d_4$ 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 $c_1, \cdots, c_4$ occupy, respectively, the starting slots of $d_1, \cdots, d_4$ 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 $c_1, \cdots, c_4$. A graphical representation of ψ_1's operation is given in Fig. 4. The arrows in Fig. 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. Fig. 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 extends randomly from d_1 either right or left one column, then proceeds vertically to c_1's row, and then horizontally one column to c_1's location. Once the 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.

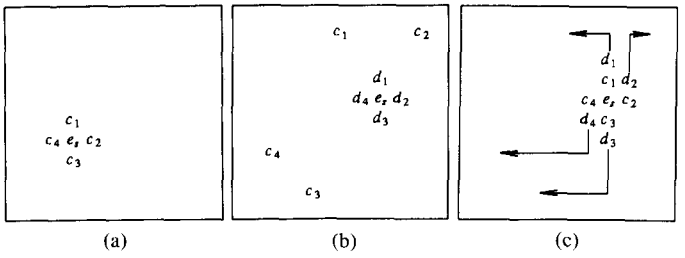

Fig. 4. Crossover operator ψ_1. (a) Passing parent. (b) Target parent. (c) Offspring.

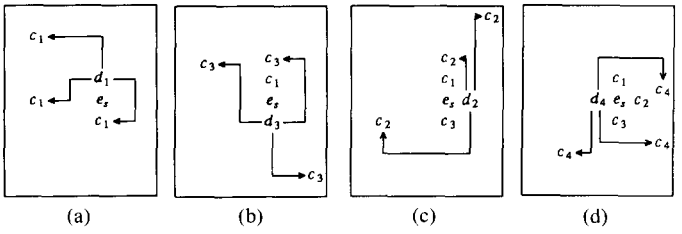

Fig. 5. Some pathological ψ_1 sliding cases and their resolution.

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. Fig. 5(b) shows several of the possible chain-forms from d_3 to c_3.

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. Fig. 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 the 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. Fig. 5(d) shows several of the possible chain-forms from d_4 to c_4.