VLSI Cell Placement Techniques

K. SHAHOOKAR AND P. MAZUMDER

Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, Michigan 48109

VLSI cell placement problem is known to be NP complete. A wide repertoire of heuristic algorithms exists in the literature for efficiently arranging the logic cells on a VLSI chip. The objective of this paper is to present a comprehensive survey of the various cell placement techniques, with emphasis on standard cell and macro placement. Five major algorithms for placement are discussed: simulated annealing, force-directed placement, min-cut placement, placement by numerical optimization, and evolution-based placement. The first two classes of algorithms owe their origin to physical laws, the third and fourth are analytical techniques, and the fifth class of algorithms is derived from biological phenomena. In each category, the basic algorithm is explained with appropriate examples. Also discussed are the different implementations done by researchers.

Categories and Subject Descriptors: B.7.2 [**Integrated Circuits**]: Design Aids—*placement and routing*

General Terms: Design, Performance

Additional Key Words and Phrases: VLSI, placement, layout, physical design, floor planning, simulated annealing, integrated circuits, genetic algorithm, force-directed placement, min-cut, gate array, standard cell

INTRODUCTION

Computer-aided design tools are now making it possible to automate the entire layout process that follows the circuit design phase in VLSI design. This has mainly been made possible by the use of *gate array* and *standard cell* design styles, coupled with efficient software packages for automatic placement and routing. Figure 1a shows a chip using the standard cell layout style, which includes some macro blocks. *Standard cells* (Figure 1b) are logic modules with a predesigned internal layout. They have a fixed height but different widths, depending on the functionality of the modules. They are laid out in rows, with *routing channels* or spaces between rows reserved for laying out the interconnects between the chip components. Standard cells are usually designed so the power and ground interconnects run horizontally through the top and bottom of the cells. When the cells are placed adjacent to each other, these interconnects form a continuous track in each row. The logic inputs and outputs of the module are available at *pins* or terminals along the top or bottom edge (or both). They are

This research was partially supported by the NSF Research Initiation Awards under the grant number MIP-8808978, the University Research Initiative program of the U.S. Army under the grant number DAAL 03-87-K-0007, and the Digital Equipment Corporation Faculty Development Award. K. Shahookar is supported by the Science and Technology Scholarship Program of the Government of Pakistan.

CONTENTS

connected by running interconnects or wires through the routing channels. Connections from one row to another are done either through vertical wiring channels at the edges of the chip or by using feed-through cells, which are standard height cells with a few interconnects running through them vertically. *Macro blocks* are logic modules not in the standard cell format, usually larger than standard cells, and placed at any convenient location on the chip.

Figure 2 shows a chip using the gate array design style. Here, the circuit consists only of primitive logic gates, such as NAND gates, not only predesigned but prefabricated as a rectangular array, with horizontal and vertical routing channels between gates reserved for interconnects. The design of a chip is then reduced to designing the layout for the interconnects according to the circuit diagram. Likewise, fabrication of a custom chip requires only the masking steps for interconnect layout.

Figure 3 shows a third chip layout style, which uses only macro blocks. These blocks may be of irregular shapes and sizes and do not fit together in regular rows and columns. Once again, space is left around the modules for wiring. For a detailed description of the layout styles, see Muroga [1982] and Ueda et al. [1986].

The placement problem can be defined as follows. Given an electrical circuit consisting of modules with predefined input and output terminals and interconnected in a predefined way, construct a layout indicating the positions of the modules so the estimated wire length and layout area are minimized. The inputs to the problem are the module description, consisting of the shapes, sizes, and terminal locations, and the *netlist*, describing the interconnections between the terminals of the modules. The output is a list of x- and y-coordinates for all modules. Figure 4 provides an example of placement, where the circuit schematic of Figure 4a is placed in the standard cell layout style in Figure 4b. Figure 4c illustrates the *Checkerboard model* of the placement in which all cells are assumed to be square and of equal size and all terminals are assumed to be at the center of the cells. Thus, the length of the interconnect from one cell to the next is one unit.

The main objectives of a placement algorithm are to minimize the total chip area and the total estimated wire length for all the nets. We need to optimize chip area usage in order to fit more functionality into a given chip area. We need to minimize wire length in order to reduce the capacitive delays associated with longer nets and speed up the operation of the chip. These goals are closely related to each other for standard cell and gate array design styles, since the total chip

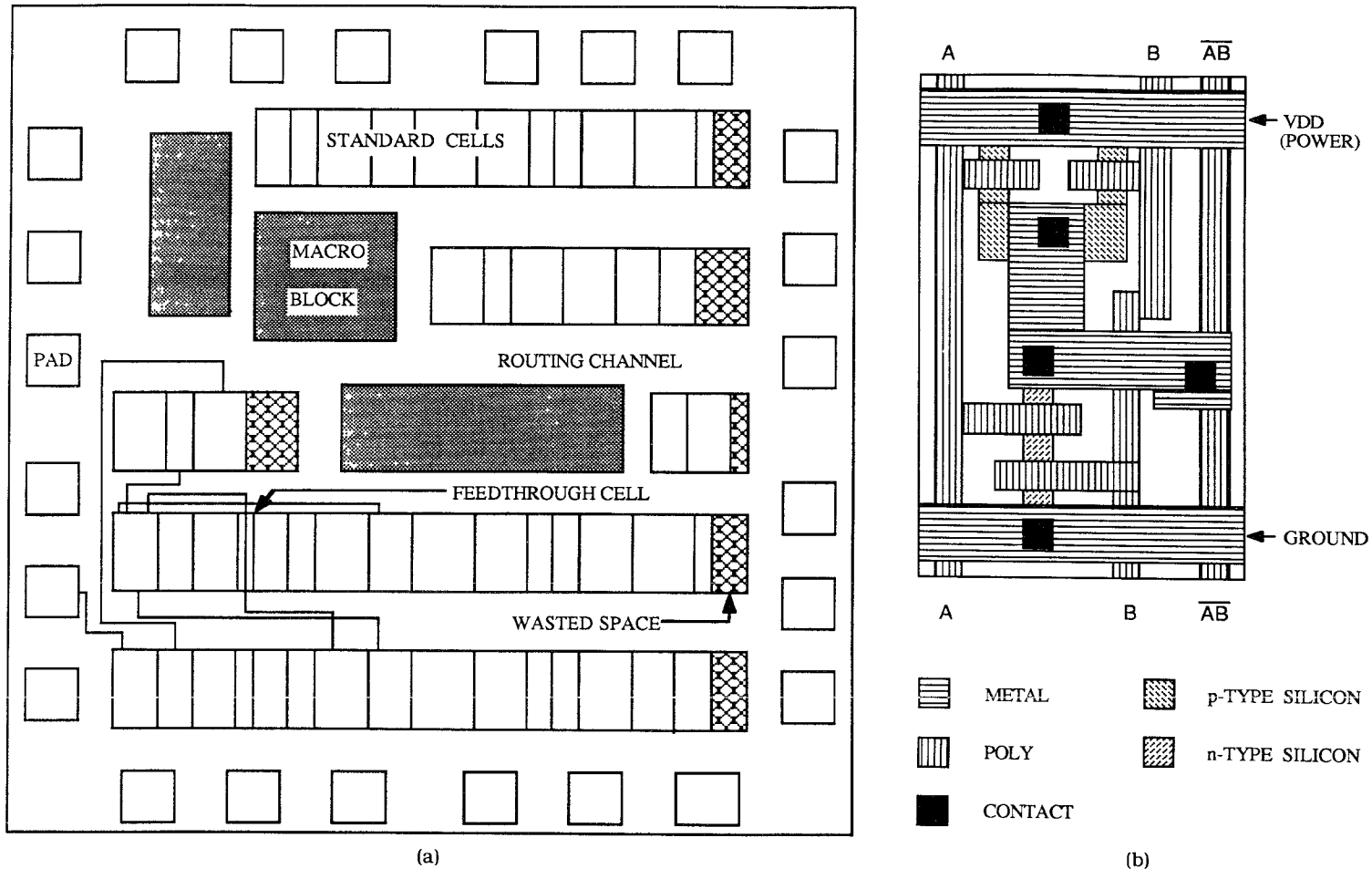


Figure 1. (a) Standard cell layout style with macro blocks. (b) Layout of a standard cell.

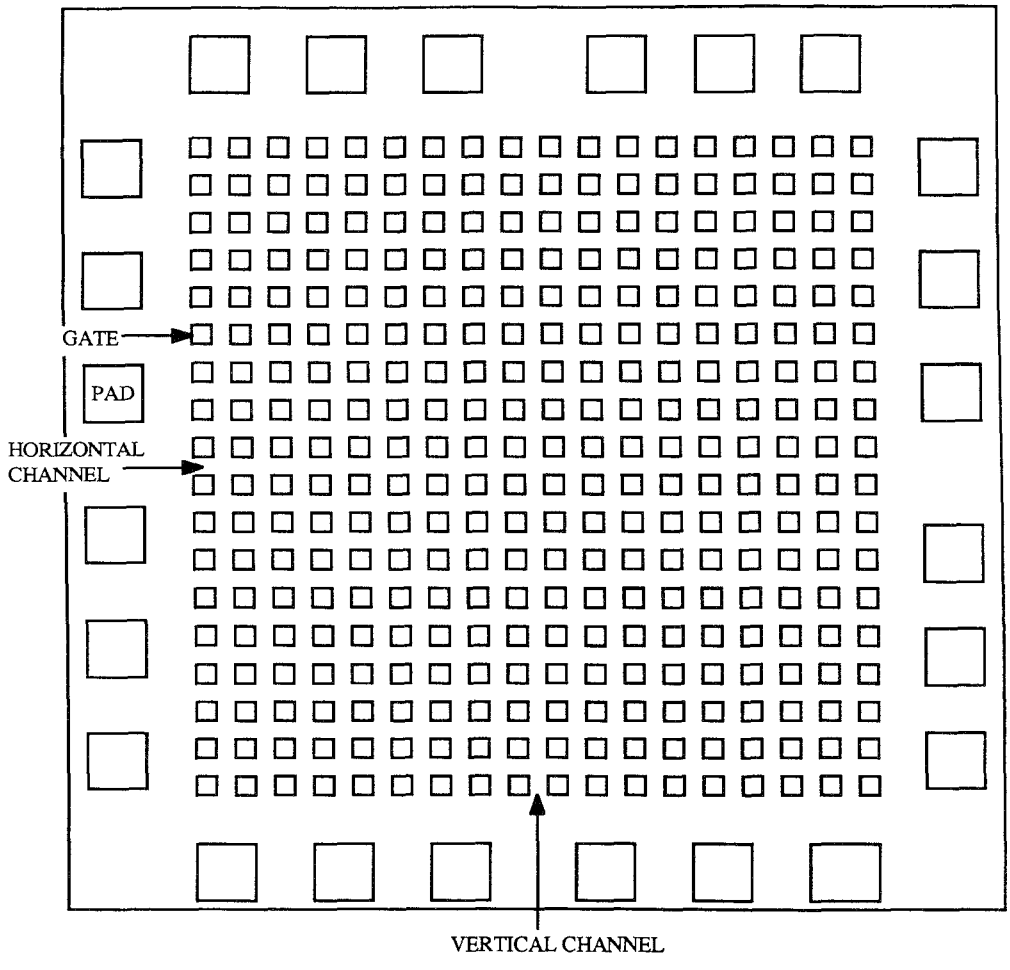


Figure 2. Gate array layout style.

area is approximately equal to the area of the modules plus the area occupied by the interconnect. Hence, minimizing the wire length is approximately equivalent to minimizing the chip area. In the macro design style, the irregularly sized macros do not always fit together, and some space is wasted. This plays a major role in determining the total chip area, and we have a trade-off between minimizing area and minimizing the wire length. In some cases, secondary performance measures, such as the preferential minimization of wire length of a few critical nets, may also be needed, at the cost of an increase in total wire length.

Another criterion for an acceptable placement is that it should be physically possible; that is, (1) the modules should not overlap, (2) they should lie within the boundaries of the chip, (3) standard cells should be confined to rows in predetermined positions, and (4) gates in a gate array should be confined to grid points. It is common practice to define a *cost function* or an *objective function*, which consists of the sum of the total estimated wire length and various penalties for module overlap, total chip area, and so on. The goal of the placement algorithm is to determine a placement with the minimum possible cost.

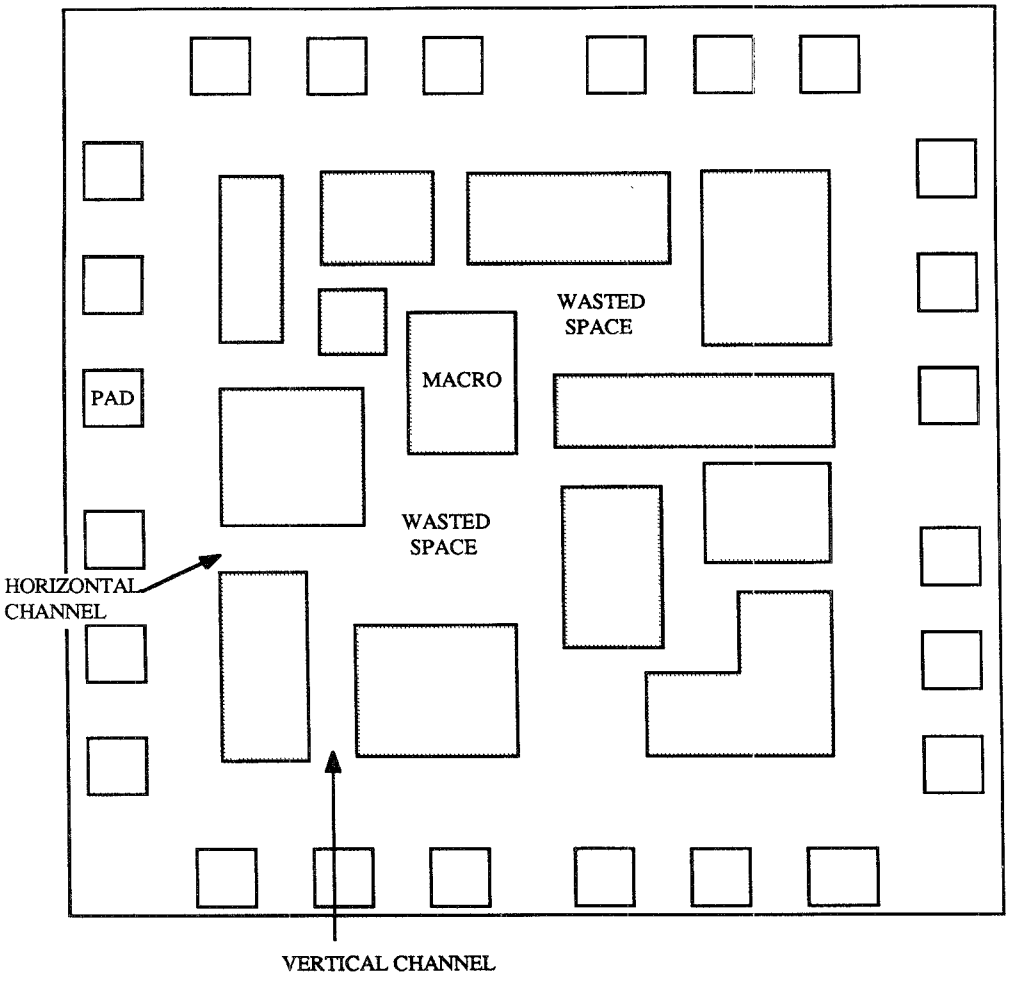
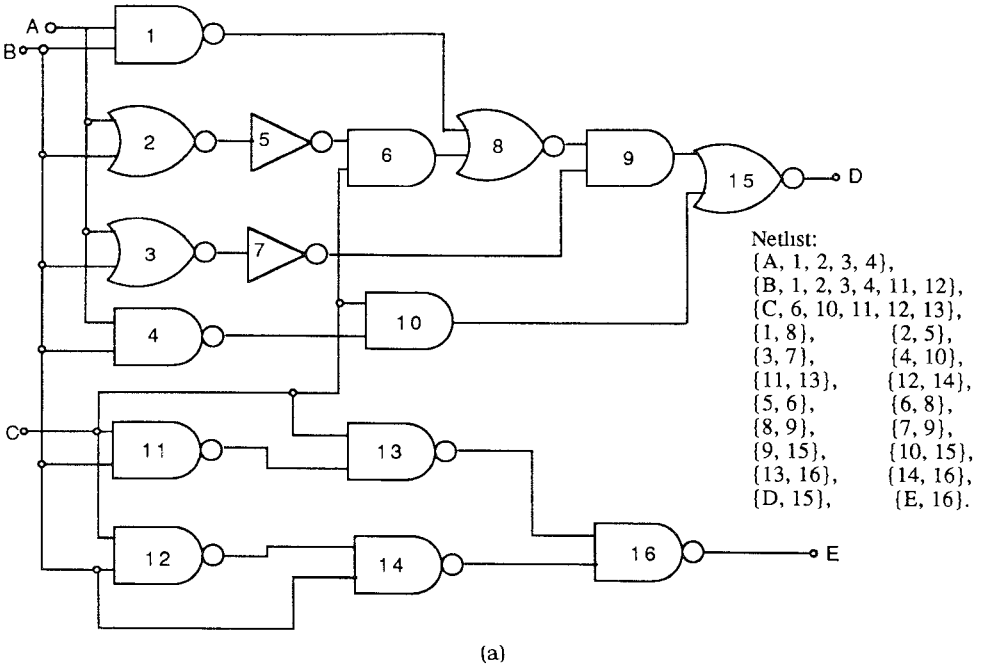


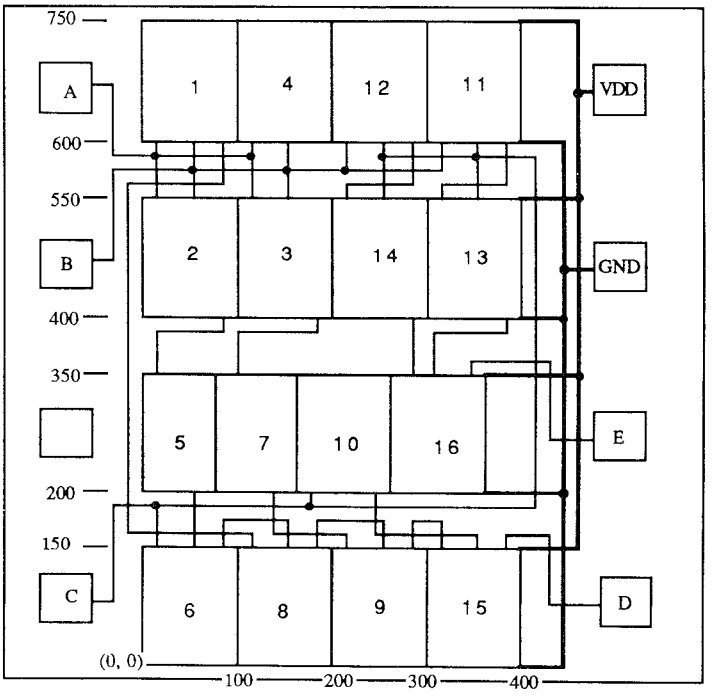
Figure 3. Macro block layout style.

Some of the placement algorithms described in this paper are suitable for standard cells and gate arrays, some are more suitable for macro blocks, and some are suitable for both. In this paper, the words *module*, *cell*, and *element* are used to describe either a standard cell or a gate (or a macro block, if the algorithm can also be used for macros). The words *macro* and *block* are used synonymously in place of macro block. Their usage also depends on the usage in the original papers. Similarly, *net*, *wire*, *interconnect*, and *signal line* are used synonymously. The terms *configuration*, *placement*, and *solution* (to the placement problem) are used synonymously to represent an assignment of modules to physical locations on the chip. The terms *pin* and *terminal* refer to terminals on the modules. The terminals of the chip are referred to as *pads*.

Module placement is an NP-complete problem and, therefore, cannot be solved exactly in polynomial time [Donath 1980; Leighton 1983; Sahni 1980]. Trying to get an exact solution by evaluating every possible placement to determine the best one would take time proportional to the factorial of the number of modules. This method is, therefore, impossible to use for circuits with any reasonable number



(a)

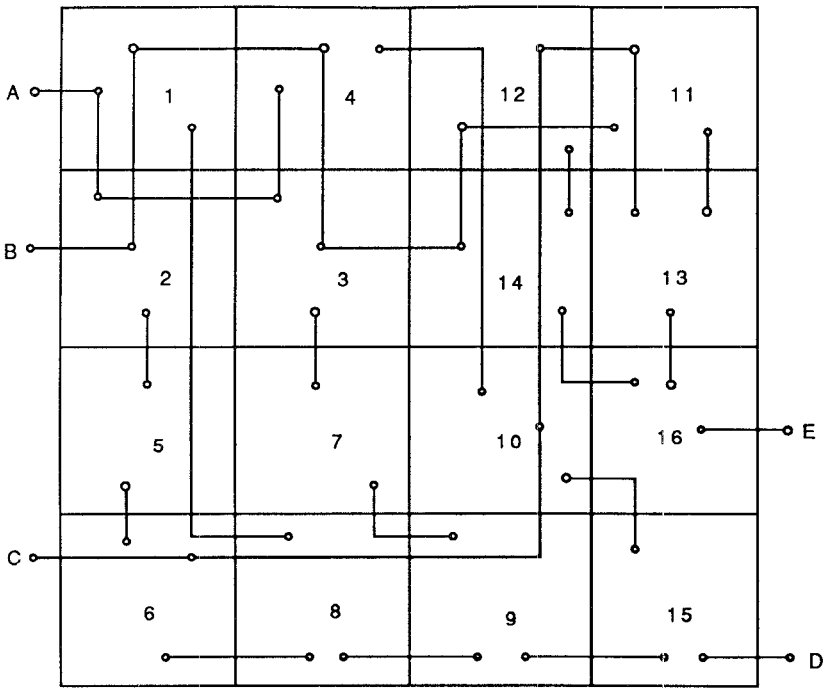


Placement
(cell, x, y):

(1, 0, 600)
(2, 0, 400)
(3, 100,400)
(4, 100,600)
(5, 0, 200)
(6, 0, 0)
(7, 75, 200)
(8, 100, 0)
(9, 200, 0)
(10, 150, 200)
(11, 300, 600)
(12, 200, 600)
(13, 300, 400)
(14, 200, 400)
(15, 300, 0)
(16, 250, 200)

(b)

Figure 4. Cell placement: problem definition. (a) Input: Netlist; (b) Output: module coordinates; (c) checkboard model.



(c)

Figure 4. Continued.

of modules. To search through a large number of candidate placement configurations efficiently, a heuristic algorithm must be used. The quality of the placement obtained depends on the heuristic used. At best, we can hope to find a good placement with wire length quite close to the minimum, with no guarantee of achieving the absolute minimum. The objective of this paper is to introduce the reader to the various heuristic algorithms developed for solving this computationally intractable problem and to analyze their performance.

The placement process is followed by *routing*, that is, determining the physical layout of the interconnects through the available space. Finding an optimal routing given a placement is also an NP-complete problem. Many algorithms work by iteratively improving the placement and, at each step, estimating the wire length of an intermediate configuration. It is not feasible to route each intermediate configuration to determine how good it is. Instead we estimate the wire length as described in the Introduction, "Wire Length Estimates."

Classification of Placement Algorithms

Placement algorithms can be divided into two major classes: *constructive placement* and *iterative improvement*. In constructive placement, a method is used to build up a placement from scratch; in iterative improvement, algorithms start with an initial placement and repeatedly modify it in search of a cost reduction. If a modification results in a reduction in cost, the modification is accepted; otherwise it is rejected.

Early constructive placement algorithms were generally based on primitive connectivity rules. For example, see Fukunaga et al. [1983], Hanan [1972a], Kambe et al. [1982], Kang [1983], Kozawa et al. [1983], Magnuson [1977], and Persky et al. [1976]. Typically, a seed module is selected and placed in the chip

layout area. Then other modules are selected one at a time in order of their connectivity to the placed modules (most densely connected first) and are placed at a vacant location close to the placed modules, such that the wire length is minimized. Such algorithms are generally very fast, but typically result in poor layouts. These algorithms are now used for generating an initial placement for iterative improvement algorithms. The main reason for their use is their speed. They take a negligible amount of computation time compared to iterative improvement algorithms and provide a good starting point for them. Palczewski [1984] discusses the complexity of such algorithms. More recent constructive placement algorithms, such as numerical optimization techniques, placement by partitioning, and a force-directed technique discussed here, yield better layouts but require significantly more CPU time.

Iterative improvement algorithms typically produce good placements but require enormous amounts of computation time. The simplest iterative improvement strategy interchanges randomly selected pairs of modules and accepts the interchange if it results in a reduction in cost [Goto and Kuh 1976; Schweikert 1976]. The algorithm is terminated when there is no further improvement during a given large number of trials. An improvement over this algorithm is *repeated iterative improvement* in which the iterative improvement process is repeated several times with different initial configurations in the hope of obtaining a good configuration in one of the trials. Currently popular iterative improvement algorithms include simulated annealing, the genetic algorithm, and some force-directed placement techniques, which are discussed in detail in the following sections.

Other possible classifications for placement algorithms are *deterministic algorithms* and *probabilistic algorithms*. Algorithms that function on the basis of fixed connectivity rules or formulas or determine the placement by solving simultaneous equations are deterministic and will always produce the same result for a particular placement problem. Probabilistic algorithms, on the other hand, work by randomly examining configurations and may produce a different result each time they are run. Constructive algorithms are usually deterministic, whereas iterative improvement algorithms are usually probabilistic.

Wire Length Estimates

To make a good estimate of the wire length, we should consider the way in which routing is actually done by routing tools. Almost all automatic routing tools use Manhattan geometry; that is, only horizontal and vertical lines are used to connect any two points. Further, two layers are used; only horizontal lines are allowed in one layer and only vertical lines in the other.

The shortest route for connecting a set of pins together is a *Steiner tree* (Figure 5a). In this method, a wire can branch at any point along its length. This method is usually not used by routers, because of the complexity of computing both the optimum branching point, and the resulting optimum route from the branching point to the pins. Instead, minimum spanning tree connections and chain connections are the most commonly used connection techniques. For algorithms that compute the Steiner tree: see Chang [1972], Chen [1983], and Hwang [1976, 1979].

Minimal spanning tree connections (Figure 5b), allow branching only at the pin locations. Hence the pins are connected in the form of the minimal spanning tree of a graph. Algorithms exist for generating a minimal spanning tree given the netlist and cell coordinates. An example of the minimal spanning tree algorithm is Kruskal [1956].

Chain connections (Figure 5c) do not allow any branching at all. Each pin is simply connected to the next one in the form of a chain. These connections are even simpler to implement than spanning tree connections, but they result in slightly longer interconnects.

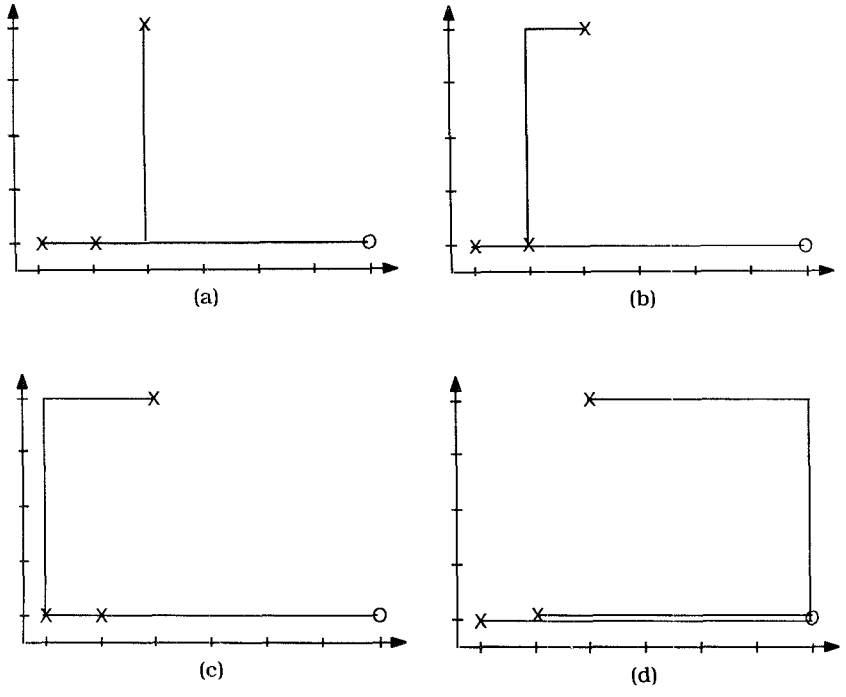


Figure 5. Some wiring schemes. (a) Steiner tree—wire length = 10; (b) minimal spanning tree—wire length = 11; (c) chain connection—wire length = 12; (d) source-to-sink connections—wire length = 19. O, source; X, sink.

Source-to-sink connections (Figure 5d), where the output of a module is connected to all the inputs by separate wires, are the simplest to implement. They, however, result in excessive interconnect length and significant wiring congestion. Hence, this type of connection is seldom used.

An efficient and commonly used method to estimate the wire length is the *semiperimeter method*. The wire length is approximated by half the perimeter of the smallest bounding rectangle enclosing all the pins (Figure 6). For Manhattan wiring, this method gives the exact wire length for all two-terminal and three-terminal nets, provided the routing does not overshoot the bounding rectangle. For four-terminal nets, in the worst case the semiperimeter estimate predicts a wire length 33% less than both the actual chain connection and spanning tree wire lengths. For nets with more pins and more zigzag connections, the semiperimeter wire length will generally be less than the actual wire length. Besides, this method provides the best estimate for the most efficient wiring scheme, the Steiner tree. The error will be larger for minimal spanning trees and still larger for chain connections. In practical circuits, however, two- and three-terminal nets are most common. Moreover, among the more complex nets, not all will be worst case, so the semiperimeter wire length is a good estimate.

Some of the algorithms described in Section 4 use the euclidean wire length or squared euclidean wire length. The squared wire length is used to save the time required for computing a square root and for floating point computations as compared to integer processing. Optimization of the squared wire length will ensure that the euclidean wire length is optimized.

1. SIMULATED ANNEALING

Simulated annealing [Kirkpatrick et al. 1983] is probably the most

well-developed method available for module placement today. It is very time consuming but yields excellent results. It is an excellent heuristic for solving any combinatorial optimization problem, such as the Traveling Salesman Problem [Randelman and Grest 1986] or VLSI-CAD problems such as PLA folding [Wong et al. 1986], partitioning [Chung and Rao 1986], routing [Vecchi and Kirkpatrick 1983], logic minimization [Lam and Delosme 1986], floor planning [Otten and van Ginnekin 1984], or placement. It can be considered an improved version of the simple random pairwise interchange algorithm discussed above. This latter algorithm has a tendency of getting stuck at local minima. Suppose, for example, during the execution of the pairwise interchange algorithm, we encounter a configuration that has a much higher cost than the optimum and no pairwise interchange can reduce the cost. Since the algorithm accepts an interchange only if there is a cost reduction and since it examines only pairwise interchanges, there is no way of progressing further from such a configuration. The algorithm is trapped at a locally optimum configuration, which may be quite poor. Experience shows that this happens quite often. To avoid such local optima, we need an algorithm that periodically accepts moves that result in a cost increase. Simulated annealing does just that.

The basic procedure in simulated annealing is to accept all moves that result in a reduction in cost. Moves that result in a cost increase are accepted with a probability that decreases with the increase in cost. A parameter T, called the *temperature*, is used to control the acceptance probability of the cost increasing moves. Higher values of T cause more such moves to be accepted. In most implementations of this algorithm, the acceptance probability is given by $\exp(-\Delta C/T)$, where ΔC is the cost increase. In the beginning, the temperature is set to a very high value so most of the moves are accepted. Then the temperature is gradually decreased so the cost increasing moves have less chance of being accepted. Ultimately, the temperature is reduced to a very low value so that only moves causing a cost reduction are accepted, and the algorithm converges to a low cost configuration.

1.1 Algorithm

A typical simulated annealing algorithm is as follows:

```
PROCEDURE Simulated_Annealing;
    initialize;
    generate random configuration;
    WHILE stopping_criterion (loop_count, temperature) = FALSE
        WHILE inner_loop_criterion = FALSE
            new_configuration  ←  perturb(configuration);
            ΔC  ←  evaluate(new_configuration, configuration);
            IF ΔC < 0  THEN  new_configuration ← configuration
            ELSE IF accept(ΔC, temperature) > random(0, 1)
                    THEN new_configuration ← configuration;
                    ENDIF
            ENDIF
        ENDWHILE
        temperature ← schedule(loop_count, temperature);
        loop_count ← loop_count + 1;
    ENDWHILE
END.
```

Perturb generates a random variation of the current configuration. This may include displacing a module to a random location, an interchange of two modules, rotation and mirroring within the restrictions of the layout geometry, or any

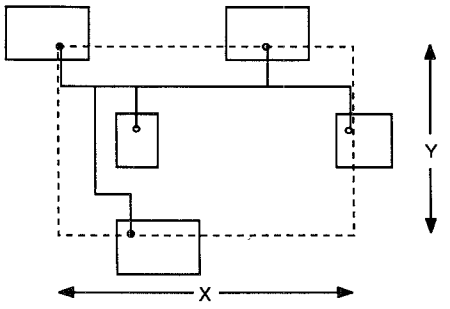


Figure 6. Semiperimeter wire length = X + Y.

other move likely to change the wire length. For standard cells, usually mirroring about the vertical axis is allowed, whereas for macro blocks, rotation in steps of 90° or mirroring about either axis is allowed. A range-limiting function may be implemented, which may first select the module to be moved, then select a destination within a specified range from the target location. This is usually done to increase the acceptance rate of the moves.

Evaluate evaluates the change in cost, using the semiperimeter method. To save CPU time, the change in wire length can be calculated incrementally. That is, the computation is done only for those nets that are connected to the cells that were moved.

Accept is the probabilistic acceptance function that is called when the cost is increased by a perturbation. It determines whether to accept a move or not, depending on the cost increase and temperature. Usually it is the exponential function described above, but it can be any other function.

Schedule is the temperature schedule, which gives the next temperature as a function of the number of iterations or the previous temperature. For example, the function $T_{i+1} = 0.1\ T_i$ may be used for exponential temperature decrease.

Inner_loop_criterion is the criterion that decides the number of trials at each temperature. Usually the number of moves attempted per cell at each temperature is fixed.

Stopping_criterion terminates the algorithm when the temperature or the number of iterations has reached a threshold value.

There are no fixed rules about the initial temperature, the cooling schedule, the probabilistic acceptance function, or the stopping criterion, nor are there any restrictions on the types of moves to be used—displacement, interchange, rotation, and so on. The quality of placement and the execution time depend on these parameters. A good choice of parameters can result in a good placement in a relatively short run time. The greatest challenge in tuning a simulated annealing algorithm lies in finding a single set of parameters and functions that consistently give very good solutions for a wide variety of circuits, while using a minimum of computation time. Initially, researchers chose these parameters and functions arbitrarily. Recently, however, several researchers have done a rigorous statistical analysis of the annealing process in order to derive more appropriate functions. Section 1.3 gives the parameters and functions used in TimberWolf, a well-known place and route package. Section 1.4 discusses other alternatives for these parameters and functions.

1.2 Operation of Simulated Annealing

If simulated annealing is run for a sufficiently long time and with the appropriate cooling schedule, it is guaranteed to converge to the global minimum [Mitra et al. 1985; van Laarhoven and Aarts 1987]. This section explains in intuitive terms why this is so. Two analogies are given to illustrate the operation of this algorithm.

In the first analogy, from which the algorithm gets its name, simulated annealing is compared to the annealing process in metals. If a metal is stressed and has imperfect crystal structure, one way to restore its *atomic placement* is to heat it to a very high temperature, then cool it very slowly. At high temperature, the atoms have sufficient kinetic energy to break loose from their current incorrect positions. As the material cools, the atoms slowly start getting trapped at the correct lattice locations. If the material

is cooled too rapidly, the atoms do not get a chance to get to the correct lattice locations and defects are frozen into the crystal structure. Similarly, in simulated annealing at high temperature, there are many random permutations in the initial configuration. These give the cells at incorrect locations a chance to get dislodged from their initial position. As the temperature is decreased, the cells slowly start getting trapped at their optimum locations.

In the second analogy, the action of simulated annealing is compared to a ball in a hilly terrain inside a box [Szu 1986]. Without any perturbation, the ball would roll downhill until it encountered a pit, where it would rest forever although the pit may be high above the minimum valley. To get the ball into the global minimum valley, the box must be shaken strongly enough so that the ball can cross the highest peak in its way. At the same time, it must be shaken gently enough so that once the ball gets into the global minimum valley it cannot get out. It must also be shaken long enough so that there is a high probability of visiting the global minimum valley. These characteristics translate directly into algorithm parameters. The strength or gentleness of the vibrations is determined by the probabilistic acceptance function and the initial temperature, and the duration of the vibrations depends on the cooling schedule and the inner loop criterion.

1.3 TimberWolf 3.2

TimberWolf, developed by Carl Sechen and Sangiovanni-Vincentelli is a widely used and highly successful place and route package based on simulated annealing. Different versions of TimberWolf have been developed for placing standard cells [Sechen 1986, 1988b; Sechen and Sangiovanni-Vincentelli 1986], macros [Cassoto et al. 1987], and floor planning [Sechen 1988a]. Version 3.2 for standard cells will be described here.

TimberWolf does placement and routing in three stages. In the first stage, the cells are placed so as to minimize the estimated wire length using simulated annealing. In the second stage, feed through cells are inserted as required, the wire length is minimized again, and preliminary global routing is done. In the third stage, local changes are made in the placement wherever possible to reduce the number of wiring tracks required. In the following discussion we will primarily be concerned with stage 1—placement. Details about the rest of the algorithm are given in Sechen [1986, 1988b] and Sechen and Sangiovanni-Vincentelli [1986].

The simulated annealing parameters used by TimberWolf are as follows.

1.3.1 Move Generation Function

Two methods are used to generate new configurations from the current configuration. Either a cell is chosen randomly and displaced to a random location on the chip, or two cells are selected randomly and interchanged. The performance of the algorithm was observed to depend upon r, the ratio of displacements to interchanges. Experimental results given in Sechen and Sangiovanni-Vincentelli [1986] indicate that the algorithm performs best with $3 \leq r \leq 8$.

Cell mirroring about the horizontal axis is also done but only when a displacement is rejected and only in approximately 10% of those cases selected at random. In addition, a temperature-dependent range limiter is used to limit the distance over which a cell can move. Initially, the span of the range limiter is twice the span of the chip, so for a range of high temperatures no limiting is done. The span decreases logarithmically with temperature:

$$L_{WV}(T) = L_{WV}(T_1)\frac{\log T}{\log T_1}$$

$$L_{WH}(T) = L_{WH}(T_1)\frac{\log T}{\log T_1}$$

where $L_{WV}(T_1)$ and $L_{WH}(T_1)$ are the desired initial values of the vertical and

horizontal window span $L_{WV}(T)$ and $L_{WH}(T)$, respectively.

1.3.2 Cost Function

The cost function is the sum of three components: the wire length cost, C_1, the module overlap penalty, C_2, and the row length control penalty, C_3.

The wire length cost C_1 is estimated using the semiperimeter method, with weighting of critical nets and independent weighting of horizontal and vertical wiring spans for each net:

$$C_1 = \sum_{\text{nets}} \left[x(i)W_H(i) + y(i)W_V(i) \right],$$

where $x(i)$ and $y(i)$ are the vertical and horizontal spans of the net bounding rectangle, and $W_H(i)$ and $W_V(i)$ are the weights of the horizontal and vertical wiring spans. Critical nets are those that need to be optimized more than the rest, or that need to be limited to a certain maximum length due to propagation delay. If they are assigned a higher weight, the annealing algorithm will preferentially place the cells connected to the critical nets close to each other in an attempt to reduce the cost. If the nets still exceed the maximum length in the final placement, their weights can be increased and the algorithm run again.

Independent horizontal and vertical weights give the user the flexibility to favor connections in one direction over the other. Thus, in double metal technology, where it is possible to stack feed throughs on top of the cells and they do not take any extra area, vertical spans may be given preference (lower weight). During the routing phase, these cells are connected using feed throughs rather than horizontal wiring spans through the channels, and precious channel space is conserved. On the other hand, in chips where feed throughs are costly in terms of area, horizontal wiring is preferred and horizontal net spans are given a lower weight. This minimizes the number of feed throughs required.

The module overlap penalty, C_2, is parabolic in the amount of overlap:

$$C_2 = W_2 \sum_{i \neq j} \left[O(i,j) \right]^2,$$

where $O(i, j)$ is the overlap between the ith and jth cell, and W_2 is the weight for this penalty. It was observed that C_2 converges to 0 for $W_2 = 1$. The parabolic function causes large overlaps to be penalized and hence discouraged more than small ones. Although cell overlap is not allowed in the final placement and has to be removed by shifting the cells slightly, it takes a large amount of computation time to remove overlap for every proposed move. Recall that wire length is computed incrementally. If too many cells are shifted in an attempt to remove overlap, it would take too much computation to determine the change in wire length. This is why most algorithms allow overlap during the annealing process but penalize it. Overlap only causes a slight error in the estimated wire length. As long as the overlap is small, this error will be small. In addition, small overlaps tend to get neutralized over several iterations. Thus, it is advantageous to penalize large overlaps more heavily than small overlaps by using a quadratic function.

The row length control penalty C_3 is a function of the difference between the actual row length and the desired row length. It tends to equalize row lengths by increasing the cost if the rows are of unequal lengths. Unequal row lengths result in wasted space, as shown in Figure 1a. The penalty is given by

$$C_3 = W_3 \sum_{\text{rows}} | L_R - \hat{L}_R |,$$

where L_R is the actual row length, $\hat{L}_R$ is the desired row length, and W_3 is the weight for this penalty for which the default value of 5 is used. Experiments show that the function used provides good control, with final row lengths within 3–5% of the desired value. Results of two experiments are given by Sechen and Sangiovanni-Vincentelli [1986], showing a reduction in wire length when the row length control penalty was introduced.

1.3.3 Inner Loop Criterion

At each temperature, a fixed number of moves per cell is attempted. This number

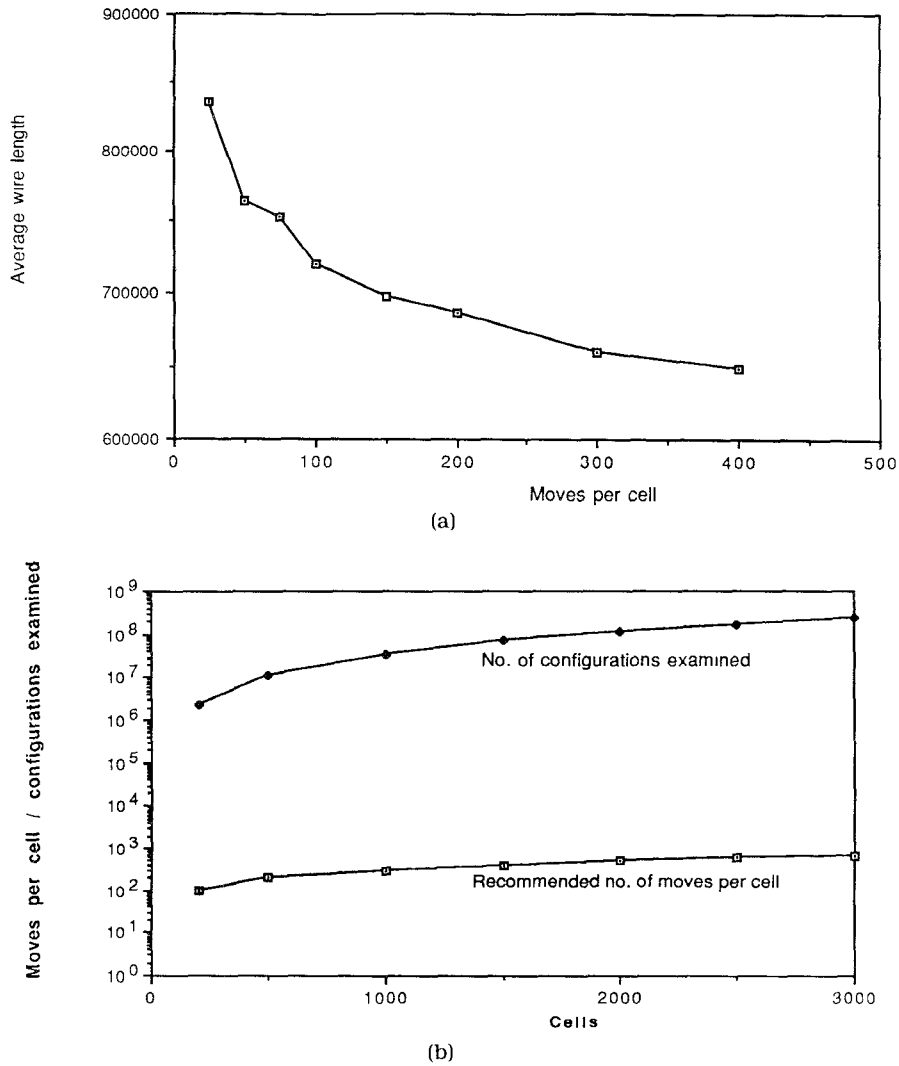


Figure 7. (a) Quality versus CPU time tradeoff in TimberWolf. (b) Recommended number of moves per cell

is a parameter specified by the user. The final wire length decreases monotonically as the number of moves per cell is increased. As the number of moves grows, however, the reduction in the final wire length diminishes, and large increases in CPU time are incurred. The optimal number of moves per cell depends on the size of the circuit. For example, for a 200-cell circuit, 100 moves per cell are recommended in Sechen [1986], which calls for the evaluation of a total of 2.34 million configurations (in 117 temperature steps). For a 3000-cell circuit, 700 moves per cell are recommended, which translates to a total of 245.7 million attempts. Figure 7a shows the final wire length as a function of the number of moves per cell for a 1500-cell problem. As the number of moves per cell is increased beyond the recommended point, the curve flattens out, thus causing little further improvement with tremendous increases in computation. Figure 7b

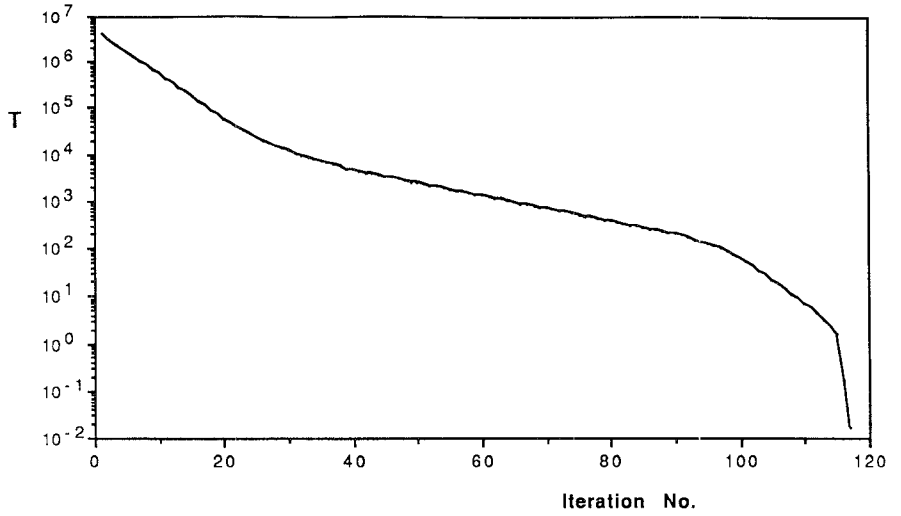


Figure 8. TimberWolf 3.2 cooling schedule.

shows the recommended number of moves per cell as a function of the problem size.

1.3.4 Cooling Schedule and Stopping Criterion

The cooling schedule can be explained by an analogy to the process of crystallization. To achieve a perfect crystal structure, it is important that around the melting point the temperature is reduced very slowly. The annealing process is started at a very high temperature, $T_1 = 4{,}000{,}000$, so most of the moves are accepted. The cooling schedule is represented by

$$T_{i+1} = \alpha(T)T_i,$$

where $\alpha(T)$ is the cooling rate parameter, which is determined experimentally. At first, the temperature is reduced rapidly [$\alpha(T) \approx 0.8$]. Then, in the medium temperature range, the temperature is reduced slowly [$\alpha(T) \approx 0.95$]. Most processing is done in this range. In the low temperature range, the temperature is reduced rapidly again [$\alpha(T) \approx 0.8$]. The resulting cooling schedule is shown in Figure 8. The algorithm is terminated when $T < 0.1$. This consists of 117 temperature steps.

1.3.5 Performance

Figure 9 shows a typical optimization curve. In the first few iterations there is so much random perturbation that the cost increases. During the first half of the run, there is almost no improvement. This perturbation is necessary to avoid entrapment at local optima. When the temperature is reduced, the cost begins to decrease. The performance of TimberWolf was compared to a commercially developed placement program based partly on the min-cut algorithm. TimberWolf achieved 17–37% smaller wire length for industrial circuits ranging from 469 to 2500 cells. The 2500-cell circuit required 15 hours of CPU time on an IBM 3081K. Compared to manual layout for an 800-cell circuit, TimberWolf achieved a 24% reduction in wire length using 4 h of CPU time on an IBM 3081K.

1.4 Recent Improvements in Simulated Annealing

Recently researchers have begun to analyze the performance of the algorithm and control its operating parameters using statistical techniques. A tenfold speedup has been reported compared with previous versions.

1.4.1 Effect of Probabilistic Acceptance Functions

Nahar, Sahni, and Shragowitz [1985] experimented with the 20 different probabilistic acceptance functions and temperature schedules listed here. In the list, g_k is the acceptance function, C_i and C_j are the previous and new costs, and T_k is the kth temperature step.

(1) Metropolis	$g_1 = \exp[-(C_j - C_i)/T_1]$
(2) Six temperature Metropolis	$g_k = \exp[-(C_j - C_i)/T_k];\quad 1 \le k \le 6$
(3) Constant	$g_1 = 1$
(See Nahar [1985] for the details of implementation of this function.)	
(4) Unit step	$g_1 = 1 \quad \text{and} \quad g_2 = 0.5$
(5) Linear	$g_1 = T_1 C_i$
(6) Quadratic	$g_1 = T_1 C_i^2$
(7) Cubic	$g_1 = T_1 C_i^3$
(8) Exponential	$g_1 = \dfrac{\exp(C_i/T_1) - 1}{e - 1}$
(9) Six temperature linear	$g_k = T_k C_i;\quad 1 \le k \le 6$
(10) Six temperature quadratic	$g_k = T_k C_i^2;\quad 1 \le k \le 6$
(11) Six temperature cubic	$g_k = T_k C_i^3;\quad 1 \le k \le 6$
(12) Six temperature exponential	$g_k = \dfrac{\exp(C_i/T_k) - 1}{e - 1};\quad 1 \le k \le 6$
(13) Linear difference	$g_1 = T_1/(C_j - C_i)$
(14) Quadratic difference	$g_1 = T_1/(C_j - C_i)^2$
(15) Cubic difference	$g_1 = T_1/(C_j - C_i)^3$
(16) Exponential difference	$g_1 = \dfrac{\exp[T_1/(C_j - C_i)] - 1}{e - 1}$
(17) Six temperature linear difference	$g_k = T_k/(C_j - C_i);\quad 1 \le k \le 6$
(18) Six temperature quadratic difference	$g_k = T_k/(C_j - C_i)^2;\quad 1 \le k \le 6$
(19) Six temperature cubic difference	$g_k = T_1/(C_j - C_i)^3;\quad 1 \le k \le 6$
(20) Six temperature exponential difference	$g_k = \dfrac{\exp[T_k/(C_j - C_i)] - 1}{e - 1};\quad 1 \le k \le 6$

For the unit step function and the six temperature functions, equal computation time was given to each step.

These functions were tried on the *Net Optimal Linear Arrangement problem*, which is the one-dimensional equivalent of the cell placement problem. All functions were given equal computation time, and the reduction in cost was compared. The results are shown in Figure 10. The figure also shows a comparison with the heuristics of Goto [1977] and Cohoon and Sahni [1983]. The best performance was exhibited by the six temperature annealing, constant, and cubic difference functions.

1.4.2 Statistical Control of Annealing Parameters

If we have a method for deriving the cooling schedule parameters by a statistical analysis of the problem itself, then the cooling schedule, instead of being fixed, can be adapted for each problem to be solved, and the annealing can proceed rapidly. Such approaches are termed *adaptive* simulated annealing algorithms. Aarts et al. [1985, 1986] and van Laarhoven and Aarts [1987] use the theory of Markov chains to derive the

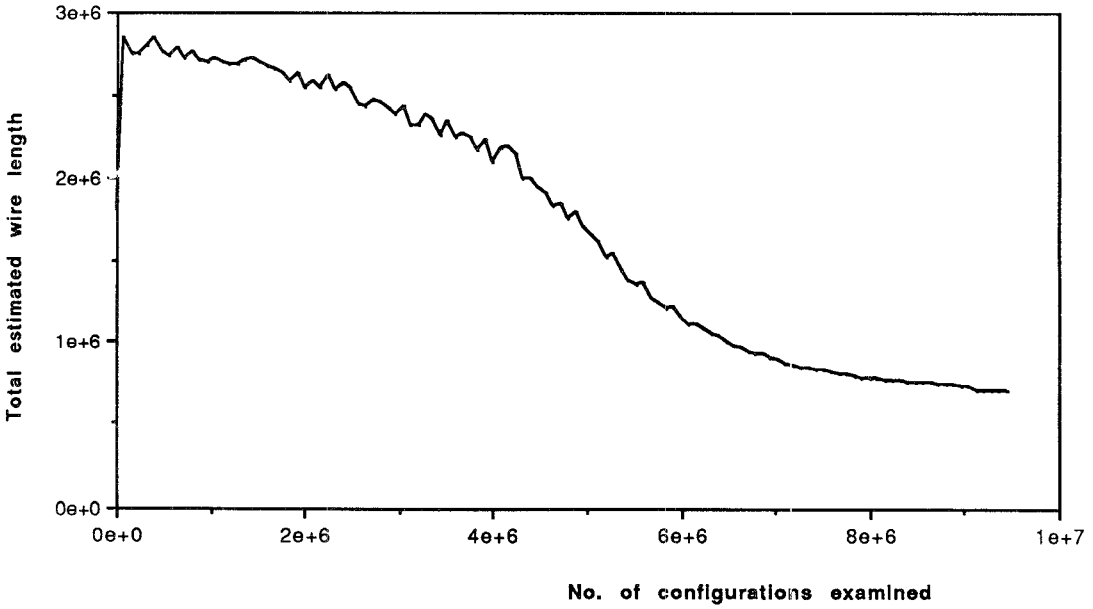


Figure 9. Optimization curve for TimberWolf 3.2.

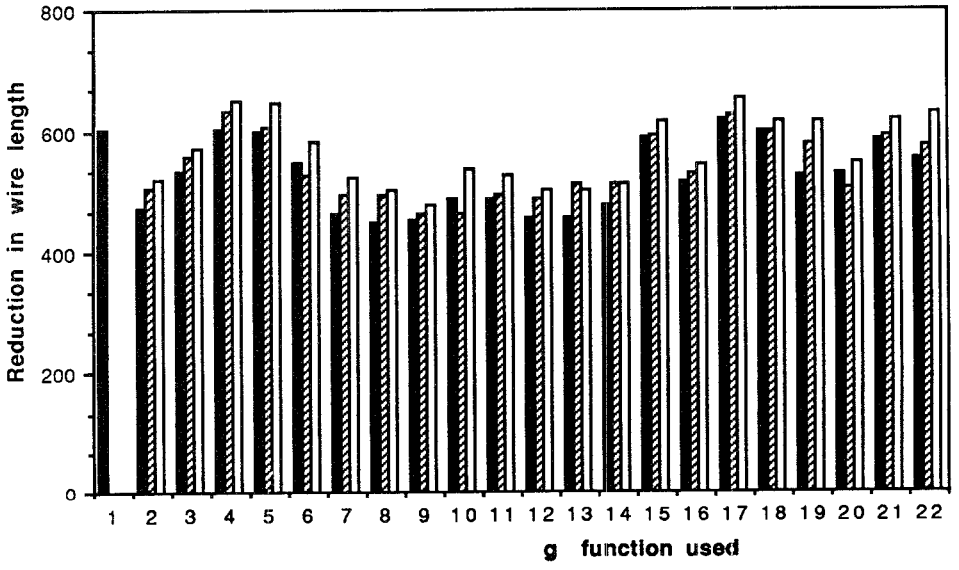


Figure 10. Comparison of various acceptance functions. ■, 6 sec; ▨, 9 sec; □ 12 sec.

cooling schedule. Similar expressions were developed by Huang et al. [1986].

Notation

$\mathbf{R} = \{\mathbf{r}_1, \mathbf{r}_2, \ldots, \mathbf{r}_{|\mathbf{R}|}\}$ is the *configuration space*, the set of all possible placements, where

i is a configuration label, which identifies a configuration uniquely,

$\mathbf{r}_i$ is the ith configuration vector, giving the coordinates of all modules in the ith placement,

$\mathbf{e}_i$ is the ith unit vector in $[0,1]^{|\mathbf{R}|}$

$I_{\mathbf{R}} = \{i \mid \mathbf{r}_i \in \mathbf{R}\} = \{1, 2, \ldots, i, \ldots, |\mathbf{R}|\}$ is the set of configuration labels,

$C: \mathbf{R} \to \mathbb{R}$ is the cost function, which assigns a real number $C(\mathbf{r}_i)$ to each configuration $i \in I_{\mathbf{R}}$ such that the lower the value of C, the better the corresponding configuration.

The algorithm can be formulated as a sequence of Markov chains, each chain consisting of a sequence of configurations for which the transition probability from configuration i to configuration j is given by

$$\Theta_{ij}(T) = \begin{cases} A_{ij}(T) P_{ij} & \text{if } i \neq j \\ 1 - \sum_{\substack{k=1 \\ k \neq i}}^{|\mathbf{R}|} A_{ik}(T) P_{ik}(T) & \text{if } i = j, \end{cases}$$

where P_{ij} is the perturbation probability, that is, the probability of generating a configuration j from configuration i (independent of T); $A_{ij}(T)$ is the acceptance probability, that is, the probability of accepting configuration j if the system is in configuration i; and T is the temperature.

The perturbation probability is chosen as

$$P_{ij} = \begin{cases} \dfrac{1}{|\mathbf{R}_i|} & \text{if } j \in I_{\mathbf{R}_i} \\ 0 & \text{if } j \notin I_{\mathbf{R}_i}, \end{cases}$$

where $\mathbf{R}_i$ is the configuration subspace for configuration i, which is defined as the space of all configurations that can be reached from configuration i by a single perturbation. This is a uniform probability distribution for all configurations in the subspace.

The acceptance probability is chosen as

$$A_{ij}(T) = \begin{cases} \exp\left(\dfrac{-\Delta C_{ij}}{T}\right) & \text{if } \Delta C_{ij} > 0 \\ 1 & \text{if } \Delta C_{ij} \leq 0, \end{cases}$$

where $\Delta C_{ij} = C(\mathbf{r}_j) - C(\mathbf{r}_i)$. This expression is known as the Metropolis criterion.

From the theory of Markov chains it follows that there exists a unique equilibrium vector $\mathbf{q}(T) \in [0, 1]^{|\mathbf{R}|}$ that satisfies

$$\text{for all } i \in I_{\mathbf{R}}: \quad \lim_{L \to \infty} \mathbf{e}_i^T \Theta(T) = \mathbf{q}^T(T).$$

If we start from any configuration, i, and perform L perturbations, with $L \to \infty$, then the probability of ending up in state j is given by the component $q_j(T)$ of the equilibrium vector. Thus, the equilibrium vector $\mathbf{q}(T)$ gives the probability distribution for the occurrence of each state at equilibrium. For the values of P_{ij} and $A_{ij}(T)$ given above,

$$q_j(T) = q_0(T) \exp\left(\frac{-\Delta C_{i_0 j}}{T}\right),$$

where i_0 is the label of an optimal configuration and $q_0(T)$ is a normalization factor given by

$$q_0(T) = \frac{1}{\sum_{k=1}^{|\mathbf{R}|} \exp\left(-\dfrac{\Delta C_{i_0 k}}{T}\right)}.$$

Further,

$$\lim_{\substack{T \to 0^- \\ L \to \infty}} \left(\mathbf{e}_i \Theta(T)\right)_j = \lim_{T \to 0^-} q_j(T) = \begin{cases} |\mathbf{R}_0|^{-1} & \text{if } j \in I_{\mathbf{R}_0} \\ 0 & \text{if } j \notin I_{\mathbf{R}_0}, \end{cases}$$

where $\mathbf{R}_0$ is the set of optimal configurations, that is, $\mathbf{R}_0 = \{\mathbf{r}_i \in \mathbf{R} \mid C(\mathbf{r}_i) = C(\mathbf{r}_{i_0})\}$. Thus, for Markov chains of infinite length, the system will achieve one of the optimal configurations with a uniform probability distribution, and the probability of achieving a suboptimal configuration is zero.

Initial Temperature. A fixed initial temperature T_1 is not used. Instead, the initial temperature is set so as to achieve a desired initial acceptance probability, χ_0. If m_1 and m_2 are the number of perturbations so far that result in cost reduction and cost increase, respectively, and if the m_2 cost-increasing perturbations are accepted according to the Metropolis criterion, the total number of configurations accepted is $m_1 + m_2 \exp(-\Delta C/T)$. This gives χ_0 as

$$\chi_0 = \frac{m_1 + m_2 \exp(-\Delta C/T)}{m_1 + m_2}.$$

This equation can be rewritten to calculate the initial temperature from the desired value of χ_0:

$$T_1 = \overline{\Delta C^{(+)}}\left[\ln\left(\frac{m_2}{m_2\chi_0 - (1-\chi_0)m_1}\right)\right]^{-1},$$

where $\overline{\Delta C^{(+)}}$ is the average value of all increases in cost, ignoring cost reductions. The initial system is monitored during a number of perturbations before the actual optimization process begins. Starting with $T_1 = 0$, after each perturbation a new value of T_1 is calculated from the above expression.

According to Huang et al. [1986], the system is considered hot enough when $T \gg \sigma$, where σ is the standard deviation of the cost function. Hence the starting temperature is taken as $T_1 = k\sigma$, where $k = -3/\ln(P)$. This allows the starting temperature T to be high enough to accept with probability P a configuration whose cost is 3σ worse than the present one. A typical value of k is 20 for $P = 0.9$. First, the configuration space is explored to determine the standard deviation of the cost function; then the starting temperature is calculated.

Temperature Decrement. Most other implementations used predetermined temperature decrements, which are not optimal for all circuit configurations. Such a cooling schedule leads to variable length Markov chains. Aarts et al. [1986] recommend the use of fixed length Markov chains. This can be achieved using the following temperature decrement:

$$T_{i+1} = T_i\left(1 + \frac{\ln(1+\delta)T_i}{3\sigma_i}\right)^{-1},$$

where σ_i is the standard deviation of the cost function up to the temperature T_i, and δ is a small real number that is a measure of how close the equilibrium vectors q_0 of two successive iterations are to each other:

$$\frac{1}{1+\delta} \le \frac{q_0(T_i)}{q_0(T_i)} \le 1 + \delta.$$

Huang et al. [1986] use the average cost versus log-temperature curve to guide the temperature decrease so that the cost decreases in a uniform manner. Hence,

$$T_{i+1} = T_i \exp\left(\frac{T_i \Delta C}{\sigma^2}\right).$$

This equation has been derived by equating the slope of the annealing curve to σ^2/T^2. To maintain quasiequilibrium, the decrease in cost must be less than the standard deviation of the cost. For $\Delta C = -\lambda\sigma$, $\lambda \le 1$,

$$T_{i+1} = T_i \exp\left(-\frac{\lambda T_i}{\sigma}\right).$$

Typically, $\lambda = 0.7$. The ratio T_{i+1}/T_i is not allowed to go below a certain lower bound (typically 0.5) in order to

prevent a drastic reduction in temperature caused by the flat annealing curve at high temperature.

Stopping Criterion. The stopping criterion is given by Aarts et al. [1986] as

$$\left| \frac{\partial \overline{C_s}(T_t)}{\partial T} \frac{T_t}{\overline{C}(T_1)} \right| < \epsilon_s,$$

where ϵ_s is a small positive number called the stop parameter, and $\overline{C}(T_1)$ is the average value of the cost function at T_1. This condition is based on extrapolating the smoothed average cost $\overline{C_s}(T_t)$ obtained during the optimization process. This average is calculated over a number of Markov chains in order to reduce the fluctuations of $\overline{C}(T_t)$.

Run-Time Complexity and Experimental Results. The Aarts et al. [1986] algorithm has a complexity $O(|\mathbf{R}| \ln |\mathbf{R}|)$, where $|\mathbf{R}|$ originates from the length of the Markov chains, and the term $\ln |\mathbf{R}|$ is an upper bound for the number of temperature steps. The perturbation mechanism can be carefully selected so that the size of configuration subspaces is polynomial in the number of variables of the problem. Consequently, the simulated annealing algorithm can always be designed to be of polynomial time complexity in the number of variables.

The Huang et al. [1986] algorithm has been tested on circuits of size 183–800 cells. It results in 16–57% saving in CPU time compared to TimberWolf for approximately the same placement quality. CPU times reported are of the order of 9 h on a VAX 11/780 for an 800-cell circuit, whereas the same circuit requires 11 h with TimberWolf 3.2.

1.4.3 Improved Annealing Algorithm in TimberWolfSC 4.2

Sechen and Lee [1987] implemented a fast simulated annealing algorithm as part of TimberWolfSC version 4.2, which is 9–48 times faster than version 3.2. As a consequence of this algorithm, placement of up to 3000 cells can be done on a Micro VAX II workstation in under 24 h of CPU time. The parameters they use are as follows.

Cost Function. The standard cost function consisting of semiperimeter wire length, with adjustable weights for vertical and horizontal nets and penalty terms for overlap and row length control has been implemented. The coding, however, is much more efficient. For example, moves that cause a large penalty are rejected without wasting CPU time on extensive wire length calculation.

Overlap Penalty. Each row is divided into nonoverlapping bins. The overlap penalty C_2 is equal to the sum of the absolute differences between the bin width, $w(b)$, and total cell width intersecting the bin, $w_c(b)$. The overlap penalty is given by $C_2 = W_2 P_O$, where the amount of overlap is given by

$$P_O = \sum_{\text{bins}} |\, w_c(b) - w(b) \,|.$$

This function can be evaluated quickly because the algorithm does not need to search through all the cells in order to determine the overlap. $w_c(b)$ is known for all bins. Whenever a cell is moved, $w_c(b)$ is updated for the bins affected.

The simulated annealing process is strongly dependent on the weight, W_2, given to this penalty in the overall cost function. Hence a negative feedback scheme has been incorporated to control this parameter dynamically as the annealing progresses:

$$W_2(i+1) = \max\left(0,\, W_2(i) + \frac{P_O - P_O^T}{\hat{L}_R}\right),$$

where P_O and P_O^T are the actual and target values of the overlap penalty, and $\hat{L}_R$ is the desired row length. This increases the penalty if the overlap is greater than the target value; otherwise

reduces it. The ideal target value of overlap has been empirically determined:

$$P_O^T = \left[1.4 - 1.15\frac{i}{i_{\max}}\right]\hat{L}_R,$$

where i is the current iteration, and $i_{\max}$ is the number of iterations (temperature values) used. This gives a target value $1.4\hat{L}_R$ at high temperature, when $i \ll i_{\max}$. As the temperature decreases, the control is tightened and the target overlap is reduced until at the final temperature it is $0.25\hat{L}_R$.

Row Length Control Penalty. A similar negative feedback dynamic control has been used for the row length control penalty function $C_3 = W_3 P_R$, where P_R gives the difference between the actual and desired row lengths. Industrial designers recommend that the maximum variation in row lengths from the desired value should be within 3%. The program tries to achieve this limit by constantly varying the weight W_3. The negative feedback control function is similar to that for the overlap penalty:

$$W_3(i+1) = \max\left(0,\ W_3(i) + \frac{P_R - P_R^T}{P_R^T}\right),$$

where P_R and P_R^T are the actual and target values of the penalty, and

$$P_R^T = \left[5 - 4\frac{i}{i_{\max}}\right]\zeta\hat{L}_R,$$

where ζ is the average row length variation. Here the initial and final values of P_R^T are

$$P_R^T(0) = 5\zeta\hat{L}_R, \quad \text{and} \quad P_R^T(i_{\max}) = \zeta\hat{L}_R.$$

Early Rejection of New Moves. While evaluating moves, the penalty is computed before the wire length. If a move incurs too much penalty, it is likely the move will be rejected. Hence there is no point in calculating the wire length for such moves. The calculation of the penalty takes a fraction of the time required for wire length computation; hence early rejection of such moves significantly reduces computation time. For early rejection, the change in penalty ΔP is computed:

$$\Delta P = \Delta C_2 + \Delta C_3 = \Delta C - \Delta C_1.$$

The acceptance probability $\exp(-\Delta C/T)$ is less than a lower limit ϵ when

$$\Delta P > |\Delta C_{1.\min}| + T|\ln \epsilon|,$$

where $\Delta C_{1\ \min}$ is the largest reduction of wire length expected in the current iteration. If the calculated penalty satisfies this inequality, the evaluation is terminated. It would be desirable to maximize the number of early rejections in order to save CPU time. This, however, also increases the number of early rejection errors—moves that were erroneously terminated, although they should have been accepted. For this purpose, a good estimate of the expected reduction in wire length $\Delta C_{1\ \min}$ is required. If the largest value of $\Delta C_{1\ \min}$ in the previous iteration is used as the estimate, the error is quite large, since ΔC_1 fluctuates substantially from iteration to iteration. For

$$|\Delta C_{1\ \min}(i)| = |\overline{\Delta C_1(i-1)}| + 1.3\sigma(i-1),$$

the early rejection error is less than 1%, where $\overline{\Delta C_1(i-1)}$ and $\sigma(i-1)$ are the mean and standard deviation of all negative values of ΔC before iteration i. With this value of $\Delta C_{1\ \min}(i)$ and with $\epsilon \approx 1/3$, we get the inequality for the early rejection test

$$\Delta P > |\overline{\Delta C_1(i-1)}| + 1.3\sigma(i-1) + T.$$

Move Generation. The previous method of maintaining a constant ratio of displacements to interchanges has been

discontinued. The following procedure is used for move generation.

A cell is selected randomly, and a random location is selected as the destination. If the destination is vacant, a displacement is performed; otherwise an interchange is performed. A new range-limiting function has been used, which restricts the motion of a cell to its neighborhood. This has caused a dramatic improvement in the move acceptance rate, thus saving the time being wasted on evaluating moves that would be rejected.

Temperature Profile. The temperature profile is the key feature of this algorithm. The dramatic improvement in the acceptance rate of new moves due to the improved move generation function has made it unnecessary to start the algorithm at a very high temperature. The temperature profile used is

$$T_1 = 500$$
$$T_{i+1} = 0.98T_i, \qquad 1 < i < 120$$

(Compare with TimberWolf 3.2, where $T_1 = 4{,}000{,}000$.) Thus, about the same number of temperature steps are concentrated in a smaller range. The final temperature is unchanged.

Acceptance Rate Control. Due to the wide variety of the circuits to be placed, a fixed temperature schedule does not always produce an appropriate value of the rate of acceptance of new configurations. It was observed that the ideal acceptance rate was 50% in the beginning ($i = 0$) and was reduced to zero at low temperatures ($i = i_{\max}$). To achieve this acceptance rate profile, negative feedback control has been provided. The ideal acceptance rate profile is given by

$$\rho_i^T = 50\left(1 - \frac{i}{i_{\max}}\right).$$

This profile is achieved by scaling the change in cost, ΔC:

$$\Delta C' = s\,\Delta C,$$

where

$$s_{i+1} = s_i\left(1 + \frac{\rho_i - \rho_i^T}{40}\right),$$

where ρ_i and ρ_i^T are the actual and target values of the percentage acceptance rate. This changes s by 2.5% for 1% deviation in ρ_i and ρ_i^T.

The algorithm was tested on six industrial circuits and was found to be 9–48 times faster than TimberWolf 3.2, with a slightly better placement. It was also tested on the MCNC benchmarks, and the wire length obtained was 10–20% better than other algorithms. The time required to achieve this improvement, however, is not given.

Some other important contributions to cell placement by simulated annealing are Bannerjee and Jones [1986], Gidas [1985], Greene and Supowit [1984], Grover [1987], Hajek [1988], Lam and Delosme [1988], Lundy and Mees [1984], Mallela and Grover [1988], Romeo and Sangiovanni-Vincentelli [1985], Romeo et al. [1984], and White [1984].

2. FORCE-DIRECTED PLACEMENT

Force-directed placement algorithms are rich in variety and differ greatly in implementation details [Hanan and Kurtzberg [1972a]. The common denominator in these algorithms is the method of calculating the location where a module should be placed in order to achieve its *ideal* placement. This method is as follows.

Consider any given initial placement. Assume the modules that are connected by nets exert an attractive force on each other (Figure 11). The magnitude of the force between any two modules is directly proportional to the distance between the modules, as in Hooke's law for the force exerted by stretched springs, the constant of proportionality being the sum of weights of all nets directly connecting them. If the modules in such a system were allowed to move freely, they

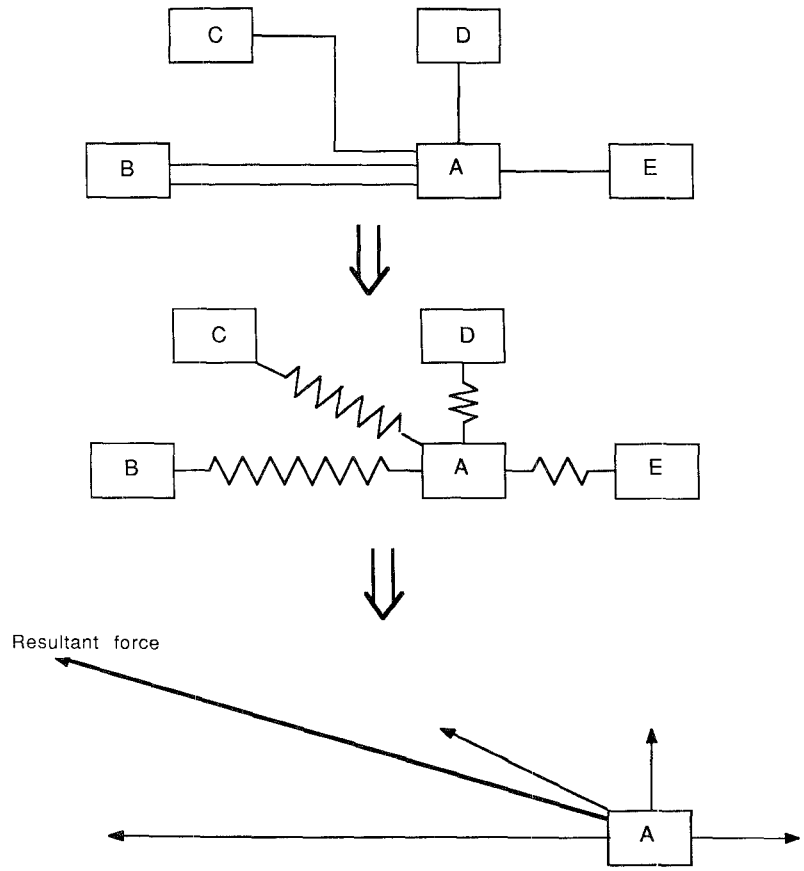


Figure 11. Force-directed placement.

would move in the direction of the force until the system achieved equilibrium in a minimum energy state, that is, with the springs in minimum tension (which is equivalent to minimum wire length), and a zero resultant force on each module. Hence the force-directed placement methods are based on moving the modules in the direction of the total force exerted on them until this force is zero.

Suppose a module M_i is connected to the module M_j by a net n_{ij} having weight w_{ij}. Let $\mathbf{s}_{ij}$ represent the distance from M_i to M_j. Then the net force on the module is given by

$$\mathbf{F}_i = \sum_j w_{ij}\mathbf{s}_{ij}.$$

If the x- and y-components of the force are equated to zero,

$$\sum_j w_{ij}(x_j - x_i) = 0,$$

$$\sum_j w_{ij}(y_j - y_i) = 0.$$

Thus, the coordinates for the zero force target point for the module M_i are given by

$$\{x_i\}_{F=0} = \frac{\sum_j w_{ij}x_j}{\sum_j w_{ij}} = \bar{x}_i,$$

$$\{y_i\}_{F=0} = \frac{\sum_j w_{ij}y_j}{\sum_j w_{ij}} = \bar{y}_i.$$

These equations resemble the center of gravity equations; that is, if the modules connected to M_i are assumed to be masses having weight w_{ij}, then the zero force target location of M_i is the center of gravity of these modules.

2.1 Force-Directed Placement Techniques

The early implementations of the force-directed placement algorithm were in the 1960s [Fisk et al. 1967]. There are many variations in existence today. Some are constructive; some are based on iterative improvement.

In constructive methods, no initial placement exists; the coordinates of each module are treated as variables, and the net force exerted on each module by all other modules is equated to zero. By simultaneously solving these equations, we get the coordinates of all modules. In such an implementation, care must be taken to avoid the trivial solution $x_i = x_j$ and $y_i = y_j$ for all i, j, which, considering the spring model, obviously satisfies the zero force condition. Another problem in this approach is that the zero force equations are nonlinear, because the force depends on distance, and the euclidean distance metric involves a square root; while the Manhattan distance metric involves absolute values. Antreich et al. [1982] give an example of the equation-solving method.

In iterative methods, an initial solution is generated either randomly or by some other constructive method. Then one module is selected at a time, its zero force target point is computed from the above equations, and an attempt is made to move the module to the target point or interchange it with the module previously occupying the target point. Such algorithms are also called force-directed relaxation or force-directed pairwise relaxation algorithms.

Here, one problem is to decide the order in which to select the modules for moving to the target location. In most implementations, the module or *seed module* with the strongest force vector is selected. In other implementations, the modules are selected randomly. In still others, the modules are selected on the basis of some estimate of their connectivity.

Another problem is where to move the selected module if the slot nearest to the zero force target location is already occupied, as it most probably will be. One solution is to move it to the nearest available free location. But the nearest free location may be very far in some cases. This is an approximate method and, at best, will need more iterations to achieve a good solution.

The second solution is to compute the target location of a module selected as described above, then evaluate the change in wire length or cost when the module is interchanged with the module at the target location. If there is a reduction in the wire length, the interchange is accepted; otherwise it is rejected. It is necessary to evaluate the wire length because it is possible that in an attempt to interchange the selected module with the module previously at the target point, we are moving that other module far away from its own target point; hence the move can result in a loss instead of a gain.

The third solution is to perform a ripple move; that is, select the module previously occupying the target point for the next move. This process is continued until the target point of a module lies at an empty slot. Then a new seed is selected.

The fourth solution is to compute the target point of each module, then look for pairs of modules such that the target point of one module is very close to the current location of the other. If such modules are interchanged, both of them will achieve their target locations with mutual benefit.

The fifth solution uses repeated trial interchanges. If an interchange reduces the cost, it is accepted; otherwise it is rejected. The cost function in this case is the sum of the forces acting on the modules. An example of the use of two types of force functions for pairwise interchange is given in Chyan and Breuer [1983].

Hanan et al. [1976a, 1976b, 1978] discuss and analyze seven placement algorithms, including three force-directed placement techniques. Experimental results are given in Hanan [1976a], and the algorithms are discussed in Hanan [1976b]. Johannes et al. [1983], Quinn [1975], and Quinn and Breuer [1979] are implementations of the force-directed algorithm.

2.2 Algorithm

Here is an algorithm for one version of the force-directed placement technique described above:

```
PROCEDURE (Force_directed_placement)
    Generate the connectivity matrix from the netlist;
    Calculate the total connectivity of each module;
    WHILE (iteration_count < iteration_limit)
        Select the next seed module, in order of total connectivity;
        Declare the position of the seed vacant;
        WHILE NOT (end_ripple)
            Compute the target point for selected module and round off to the nearest integer;
            CASE target point:
                LOCKED
                    Move selected module to nearest vacant location;
                    end_ripple ← TRUE;
                    Increment abort_count;
                    IF abort_count > abort_limit
                    THEN
                        Unlock all modules;
                        Increment iteration_count;
                    ENDIF;
                OCCUPIED:
                    Select module at target point for next move;
                    Move previous selected module to target point and lock;
                    end_ripple ← FALSE;
                    abort_count ← 0;
                SAME:
                    Do not move module;
                    end_ripple ← TRUE;
                    abort_count ← 0;
                VACANT:
                    Move selected module to target point and lock;
                    end_ripple ← TRUE;
                    abort_count ← 0;
        ENDCASE;
    ENDWHILE;
END.
```

This implementation uses ripple moves in which a selected module is moved to the computed target point; if the target point was previously occupied, the module displaced from there is selected to be moved next. When a module has been moved to its target point, it is necessary to lock it for the rest of the current iteration in order to avoid infinite loops. For example, suppose two modules, A and B, are competing for the same target location and we move A to the target location. Then we select B for the next move and compute the same target point for it. If we move B to the target location, it will displace A and we will have to compute the new target point for A, which will be the same again. Hence A and B will keep displacing each other. When the number of locked modules exceeds a

limit (depending on the size of the netlist), there will be too many aborts. At that time all modules are unlocked again, another seed is selected, and a new iteration is started.

2.3 Example

Consider a circuit consisting of nine modules, with the following netlist:

net 1 = {1 3 4 8 9}

net 2 = {1 5 6 7 8 9}

net 3 = {2 4 5 6 7 9}

net 4 = {3 7}

The lower bound on the wire length for this example is 15, assuming each hop of a net from one terminal to the next is 1 unit (e.g., net 1 must be at least 4 units in order to connect five terminals). To demonstrate how force-directed placement works, we start with a random placement with a wire length of 20, as shown in Figure 12a. Table I gives the connectivity matrix. Two iterations are shown in detail in Table 2. In the first iteration, module 9 is selected as the seed module, since it has the largest connectivity, 14. The target point is (1.1, 1), using the center of gravity formula with the entries in Table 1 as weights. Hence module 9 is moved to location (1, 1), leaving its original location (0, 1) vacant. The last column of Table 2 gives the intermediate placement. Module 8, which was previously located at (1, 1), is selected for the next move. The target point is (0.9, 0.9), but we cannot place it at (1, 1) since we already placed module 9 there. Hence, it is placed in the nearest vacant slot (0, 1). Then module 7 is selected as the seed, and the process is repeated. The final solution is shown in Figure 12b. The result is an improvement in wire length of 3 units.

2.4 Goto's Placement Algorithm

Goto proposed a somewhat unique force-directed placement algorithm [Goto 1981; Goto and Matsuda 1986]. This algorithm consists of an initial placement part and

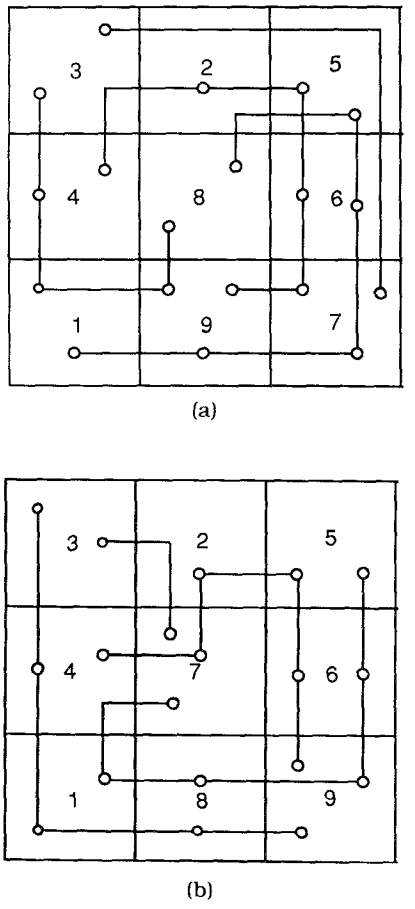


Figure 12. Force-directed placement example. (a) Random initial placement with wire length 20; (b) final placement after two iterations with wire length 17.

an iterative improvement part. The initial placement part selects modules for placement on the basis of connectivity. When selected, a module is placed at the location that yields the minimum wire length. It is not moved during the rest of the initial placement phase.

The iterative improvement part uses a generalized force-directed relaxation technique in which interchanges of two or more modules in the ϵ-neighborhood of the median of a module are explored. The *median* of a module is defined as the position at which the wire length for the nets connected to the module is minimum. The *ϵ-neighborhood* of the median

Table 1. Connectivity Matrix for the Force-Directed Placement Example

Modules	1	2	3	4	5	6	7	8	9	Σ
1	0	0	1	1	1	1	1	2	2	9
2	0	0	0	1	1	1	1	0	1	5
3	1	0	0	1	0	0	1	1	1	5
4	1	1	1	0	1	1	1	1	2	9
5	1	1	0	1	0	2	2	1	2	10
6	1	1	0	1	2	0	2	1	2	10
7	1	1	1	1	2	2	0	1	2	11
8	2	0	1	1	1	1	1	0	2	9
9	2	1	1	2	2	2	2	2	0	14

Table 2. First Two Iterations for the Force-Directed Placement Example

Iteration	Selected Module	Target Point	Case	Placed at	Result
1	9 (Seed)	(1.1, 1)	Occupied	(1, 1)	3 2 5 4 9 6 1 - 7
	8	(0.9, 0.9)	Locked	(1, 0)	3 2 5 4 9 6 1 8 7
	7 (Seed)	Locked	(1.1, 1.2)	Abort	3 2 5 4 9 6 1 8 7
	6 (Seed)	Locked	(1.2, 0.9)	Abort	3 2 5 4 9 6 1 8 7
2	9 (Seed)	(1.1, 0.9)	Same	Not moved	3 2 5 4 9 6 1 8 7
	7 (Seed)	(1.1, 1.2)	Occupied	(1, 1)	3 2 5 4 7 6 1 8 -
	9	(0.9, 1)	Locked	(2, 0)	3 2 5 4 7 6 1 8 9
	6 (Seed)	Locked	(1.2, 0.9)	Abort	3 2 5 4 7 6 1 8 9
	5 (Seed)	Locked	(1.2, 0.7)	Abort	3 2 5 4 7 6 1 8 9

of a module is defined as the set of ϵ positions for the module, where the wire length associated with it has the smallest ϵ values. Goto shows that the problem of finding the median and its ϵ-neighborhood is separable in x and y, and hence the x- and y-coordinates of the median can be calculated independently of each other using the algorithm of Johnson and Mizoguchi [1978].

The λ-*neighborhood* of a given configuration in the configuration space is defined as the set of configurations that can be obtained from the given configuration by circularly interchanging not more than λ modules. A configuration is said to be λ-*optimal* (locally optimal) if it is the best one in such a neighborhood. The process of replacing the current configuration with a better configuration from its λ-neighborhood is called *local transformation*.

The complete placement algorithm is as follows. An initial placement is generated. Generalized force-directed relaxation is performed to obtain a λ-optimum configuration. If the given amount of computation time is not exhausted, this procedure is repeated with another initial placement. The best result of all the trials is accepted. The heuristic search procedure used for finding λ-optimum configurations is now described.

The procedure consists of module interchange cycles, iterated until there is no further improvement. At the beginning of each interchange cycle, a seed module (M) is selected and interchanged on a trial basis with all modules $M(i)$ in its ϵ-neighborhood ($1 < i < \epsilon$). If there is a reduction in wire length, the interchange yielding the maximum reduction is accepted, and the interchange cycle is terminated. If there is no reduction in wire length, a triple interchange is tried between the seed module M, a module $M(i)$ in its ϵ-neighborhood, and a module $M(ij)$ in the ϵ-neighborhood of $M(i)$ ($1 < i, j < \epsilon$). This results in ϵ^2 trials in which the modules are interchanged in the cyclic order $M \rightarrow M(i) \rightarrow M(ij) \rightarrow M$. If there is a reduction in wire length, then the interchange giving the minimum wire length is accepted, and the interchange cycle is terminated. Otherwise for each i, the $j = j_i$ giving the minimum wire length is chosen for further processing. The next step is to try quadruple interchanges between M, $M(i)$, $M(ij_i)$ and the modules $M(ij_ik)$ in the ϵ-neighborhood of $M(ij_i)$ ($1 < i, k < \epsilon$). This once again results in ϵ^2 interchanges of the form $M \rightarrow M(i) \rightarrow M(ij_i) \rightarrow M(ij_ik) \rightarrow M$. We choose the k that results in the minimum wire length for further processing. This process is repeated until interchanges of λ elements have been considered. The possible interchanges are shown as a tree in Figure 13a. The interchanges that result in the minimum wire length at each step are represented by the solid lines and are pursued further, whereas those represented by the dotted lines are abandoned. There is only one solid line under any node, except the root node M.

The parameter ϵ represents the breadth of the search tree, and λ represents its depth. As ϵ and λ are increased, the λ-optimal configuration gets better, but there is also a large increase in computation time. Goto observed that ϵ = 4–5 and λ = 3–4 is the best compromise between placement quality and computation time. These results were obtained from experiments on a 151 module circuit. For satisfactory placement of larger circuits, a higher value of ϵ and λ may be necessary.

2.5 Analysis

It can be shown that the minimum energy state of the force model does not always yield the optimum wire length and vice versa. Consider the example in Figure 14a, where a module is connected by two nets to the left and by one net toward the right. The zero force position would be at a distance 10 units from the left and 20 units from the right, yielding a wire length of 40. For optimal wire length, the module should be positioned to the extreme left, yielding a wire length of only 31. Similarly, consider a module connected by one net each toward the left and right (Figure 14b). Although the module may be positioned anywhere and its x-coordinate does not affect the wire length, force-directed placement methods will unnecessarily constrain it to the center location, perhaps displacing some other module that really ought to be at that location.

Because of the inherent nature of the center of gravity formula used, force-directed methods tend to place all modules in the center of the circuit. The

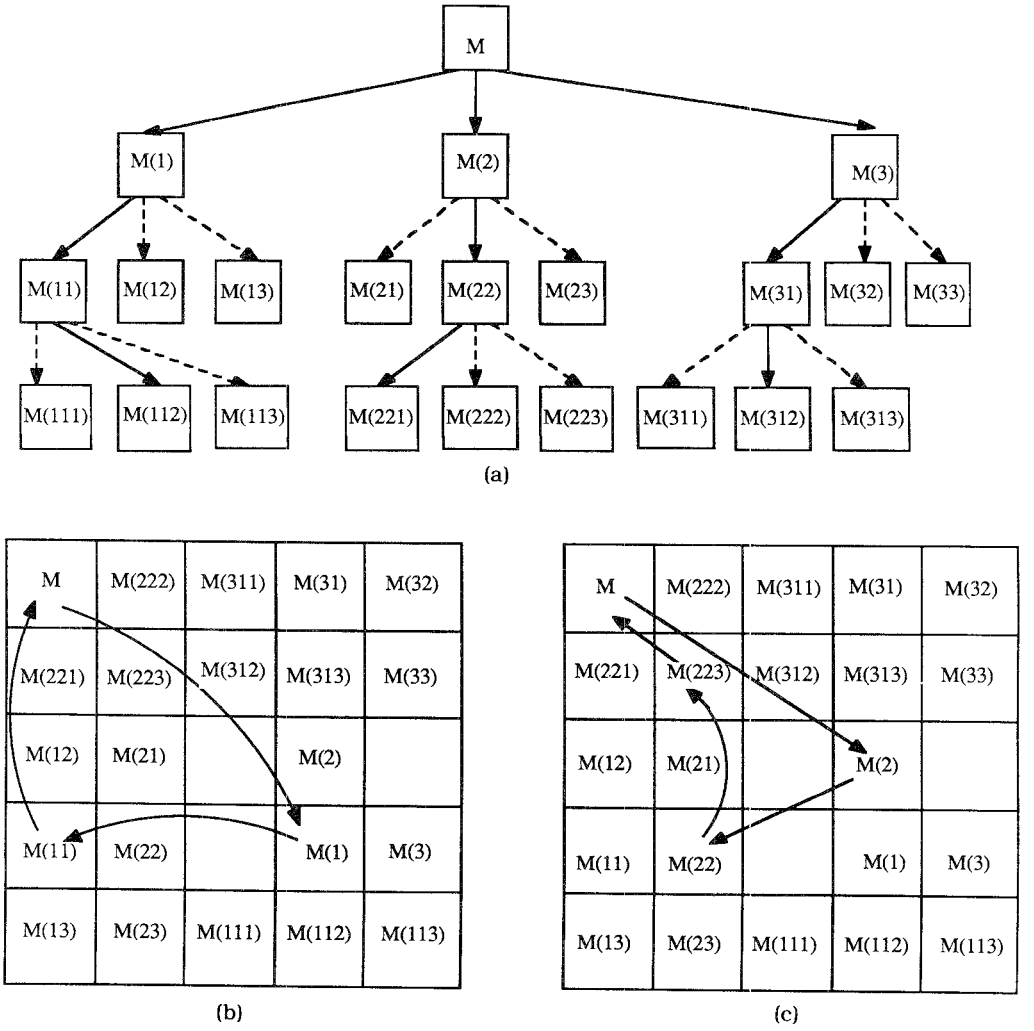


Figure 13. Force-directed relaxation. (a) Search tree; (b) exchange $\lambda = 3$; (c) exchange $\lambda = 4$.

result is too many ties and aborts, with all modules constantly displacing the center modules.

On the whole, this is a moderately good method of module placement. When fine tuned properly and combined with other strategies discussed above, it gives good results. But it is inferior in solution quality to simulated annealing.

3. PLACEMENT BY PARTITIONING

Placement by partitioning is an important class of placement algorithms based on repeated division of the given circuit into densely connected subcircuits such that the number of nets cut by the partition is minimized. Also, with each partitioning of the circuit, the available chip area is partitioned alternately in the horizontal and vertical direction (Figure 15). Each subcircuit is assigned to one partition of the chip area. If this process is carried on until each subcircuit consists of only one module, then each module will have been mapped to a unique position on the chip. Most placement by partitioning algorithms, or *Min-cut algorithms*, use some modified form of the Kernighan–Lin [1970] and

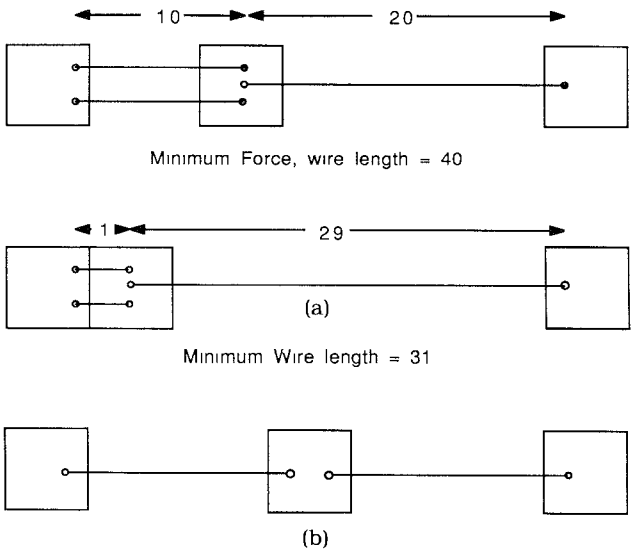


Figure 14. Problems with force-directed placement.

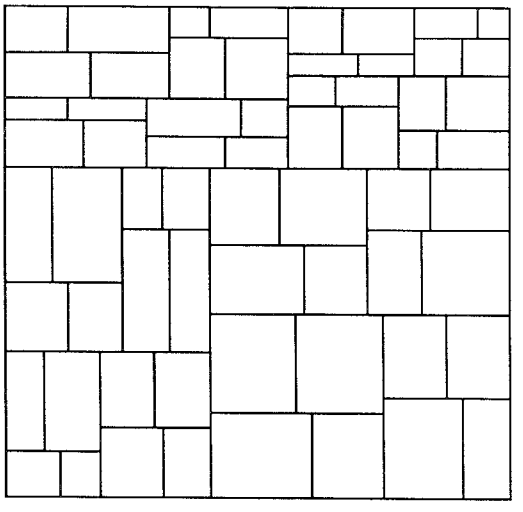


Figure 15. Chip area partitioned alternately in the vertical and horizontal direction.

Fiduccia–Mattheyses [1982] heuristics for partitioning; see also Schweikert and Kernighan [1972].

The Kernighan–Lin partitioning algorithm is as follows. Start with a random initial partition that divides the set of modules into two disjoint sets A and B. Evaluate the net cut (the number of nets connecting modules in A to modules in B and are therefore cut by the partition). For all pairs (a, b), $a \in A$, $b \in B$, find the reduction g in the net cut obtained by interchanging a and b (moving a to set B and b to A). g is called the gain of the interchange. If $g > 0$, then the interchange is beneficial. Select the module pair (a_1, b_1) with the highest gain g_1. Remove a_1 and b_1 from A and B, and find the new maximum gain g_2 for a pairwise interchange (a_2, b_2). Continue this process until A and B are empty. Find a value k such that the total gain

$$G = \sum_{i=1}^{k} g_i$$

is maximized, and interchange the corresponding module pairs (a_1, b_1), $\ldots$, (a_k, b_k). Repeat this process until $G > 0$ and $k > 0$.

Figure 16 shows an example of placement by partitioning. Figure 4 shows the circuit to be placed and the desired locations of pads. This circuit is repeatedly partitioned as shown in Figure 16. At each step, the number of nets intersected by the cut line is minimized, and the subcircuits are assigned to horizontally

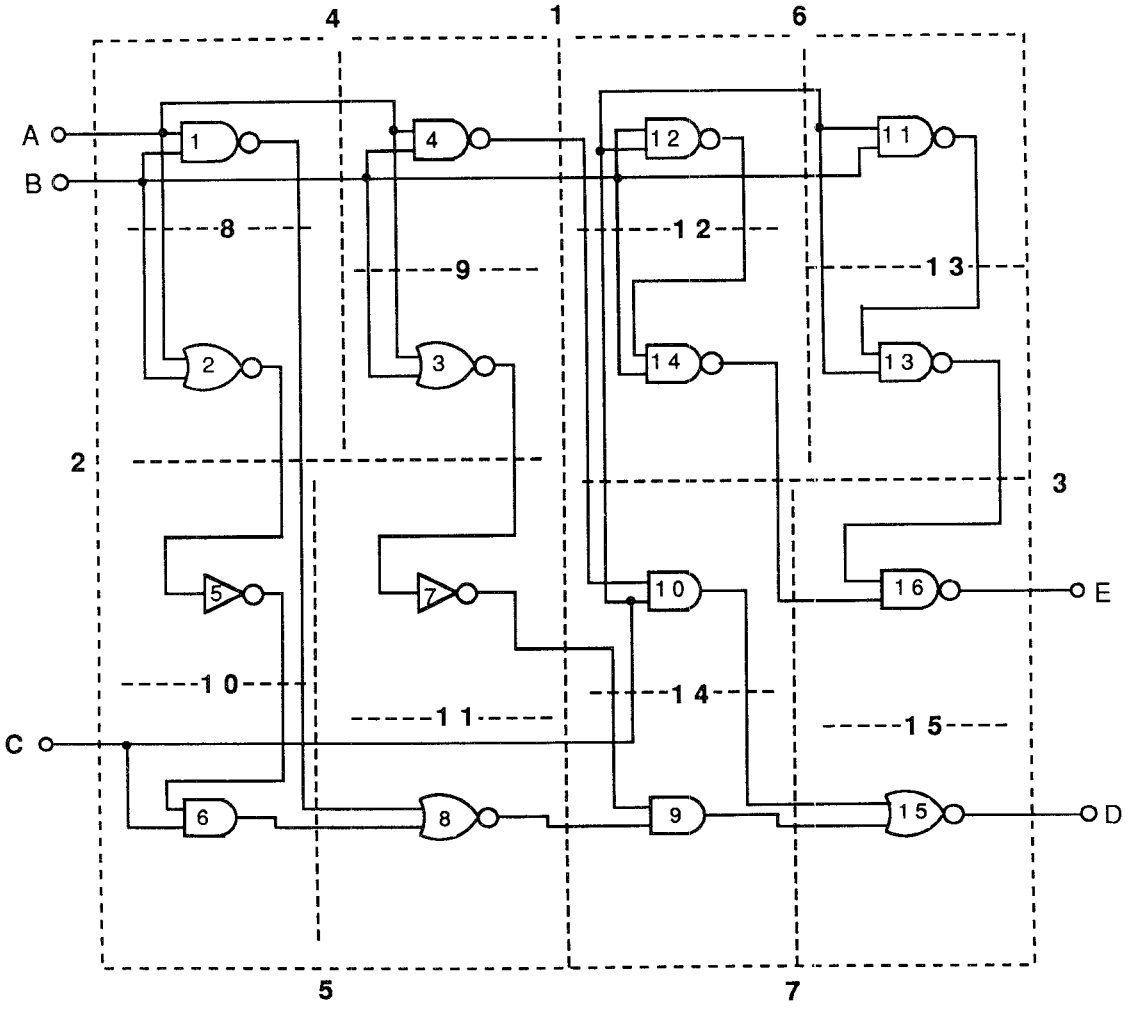


Figure 16. Min-cut partitioning of the circuit in Figure 4a.

or vertically partitioned chip areas. The resulting placement (Figure 4c) yields a total wire length of 43 (for chain connections).

3.1 Breuer's Algorithms

Breuer's algorithms [1977a, 1977b] are among the early applications of partitioning for placement. They minimize the number of nets that are cut when the circuit is repeatedly partitioned along a given set of cut lines. Consider a set of modules connected by a set of nets. Let c be a line crossing the surface of the chip. If one or more elements connected to a net s are on one side of c and one or more elements are on the other side, then, while routing the net, at least one connection must cross line c. The cut line c is said to cut the net s. For a given placement, the value of c, denoted by $v(c)$ is the total number of nets cut by c.

The following objective functions have been developed for min-cut placement:

(1) *Total net-cut.* This objective function considers the total number of nets cut by all the cut lines

partitioning the chip,

$$N_c(\sigma) = \sum v(c),$$

where the sum is over all vertical and horizontal cut lines. Consider a *canonical* set of cut lines as the collection of cut lines between each row and each column of slots. Then, minimizing the total number of nets cut using this set of cut lines is equivalent to minimizing the semiperimeter wire length. For a formal proof, see Breuer [1977a, 1977b].

(2) *Min–max cut value objective function.* In standard cell and gate array technologies, the channel width, and therefore the chip area, depend on the maximum number of nets being routed through a channel at any point or the maximum net-cut for any cut line across the channel. The form of this objective function is

$$N_c(mM) = \sum_{\text{channels}} \max_{c \in C_i} v(c),$$

where C_i is a set of cut lines defined across channel i. Note that for this objective function, only the net-cut in the congested region of the routing channel is significant, and the algorithm will try to minimize this maximum net-cut, even at the expense of increasing the net-cut in other vacant regions of the channel.

(3) *Sequential cut line objective function.* Although the above objective functions better represent the placement problem, it is computationally difficult to minimize them. A third objective function is therefore introduced, which is easy to minimize but does not give a globally optimal placement. As the name implies, the objective is to make one cut and minimize the net-cut, then to cut each group again and minimize the net-cut with respect to these cut lines and subject to the constraints already imposed by the previous cut, and so on. Note that because of the sequential (greedy) nature of this objective function, it does not guarantee that the total number of nets cut by all cut lines will be minimized. Hence, minimizing this objective function is not equivalent to minimizing the semiperimeter wire length.

3.1.1 Algorithms

Breuer has explored two basic placement algorithms. Each of these algorithms requires a given sequence of cut lines that partition the chip, so that each section contains only one slot. To be consistent with Breuer's notation, in the following discussion the subsections of the chip created by the partitioning process are called blocks. These should not be confused with macro blocks.

Cut Oriented Min-Cut Placement Algorithm. Start with the entire chip and a given set of cut lines. Let the first cut line partition the chip into two blocks. Also partition the circuit into two subcircuits such that the net-cut is minimized. Now partition all the blocks intersected by the second cut line, and partition the circuit correspondingly. Repeat this procedure for all cut lines. This process is shown in Figure 17a.

This algorithm realizes the sequential objective function described above. In practice, however, this algorithm does not always give good results because of two problems associated with it. Consider Figure 17a. While processing cut line c_2, we must partition blocks A and B created by c_1 simultaneously. First, if there is a way to partition them sequentially, computation time would be saved as a result of a reduction in the problem size. Besides, a conflict can arise when we try to bisect blocks A and B using the same cut line. If the modules of A to be placed above c_2 require a larger area than the corresponding elements in B, then it is impossible to bisect A and B with the same cut line, and a less optimal partition has to be accepted. To avoid both of these problems, another algorithm is presented in which each block is partitioned using a separate cut line.

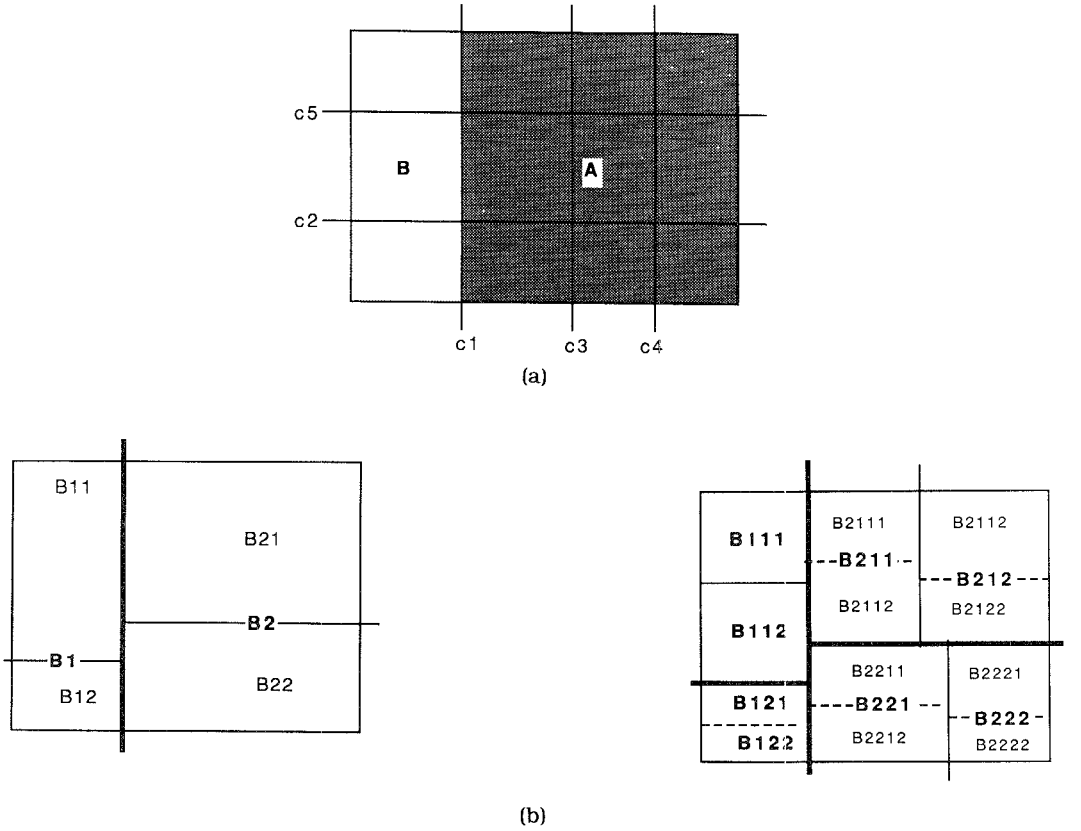


Figure 17. Breuer's min-cut algorithms. (a) Cut-oriented min-cut placement; (b) block-oriented min-cut placement.

Block-Oriented Min-Cut Placement Algorithm. In this algorithm, we select a cut line to partition the chip into two regions. Then we select a separate cut line for each region and partition the regions further. This process is repeated until each block consists of one slot only. Here, different regions can have different cut lines, as shown in Figure 17b. Note that we are no longer minimizing the sequential objective function, since we are not making uniform cuts through the entire chip.

The cut lines for partitioning the chip may be selected in any sequence. Breuer has given three sequences (Figure 18), which are most suitable for three different types of layout. These are as follows:

(1) *Quadrature Placement Procedure.* In this algorithm the partitioning process is carried out breadth first, with alternate vertical and horizontal cuts. This process is illustrated in Figure 18a. With each cut, a region is subdivided into two equal subregions. This method is suitable when there is a high routing density in the center. By first cutting through the center and minimizing the net-cut, the congestion in the center is reduced. This is currently the most popular sequence of cut lines for min-cut algorithms.

(2) *Bisection Placement Procedure.* In this procedure, the chip is repeatedly *bisected* (divided into two equal subregions) by horizontal cut lines until each subregion consists of one row. This process assigns each element to a row without fixing its position.

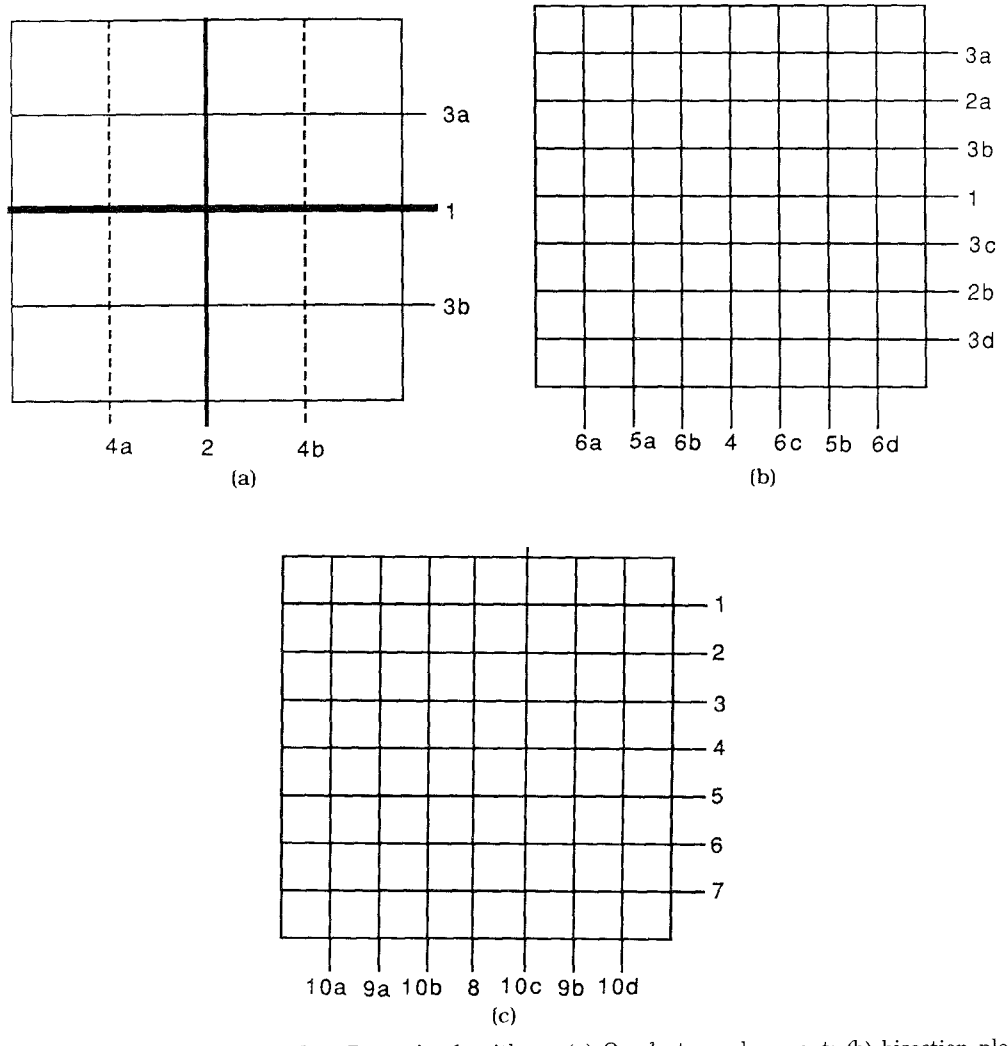


Figure 18. Cut sequences used in Breuer's algorithms. (a) Quadrature placement; (b) bisection placement; (c) slice/bisection placement.

Then each row is repeatedly bisected until each resulting subregion contains only one slot, and thus all movable modules have been placed (Figure 18b). This is a good method for standard cell placement. It does not, however, guarantee the minimization of the maximum net-cut per channel.

(3) *Slice Bisection Procedure.* Another placement strategy is to partition a suitable number of modules from the rest of the circuit and to assign them to a row (slicing) by horizontal cut lines. This process is repeated until each module has been assigned to a row. Then the modules in each row are assigned to columns by bisecting, using vertical cut lines (Figure 18c). This technique is most suitable when there is a high interconnect density at the periphery.

3.2 Dunlop's Algorithm and Terminal Propagation

When partitioning a circuit or a section of the circuit into two parts, it is not

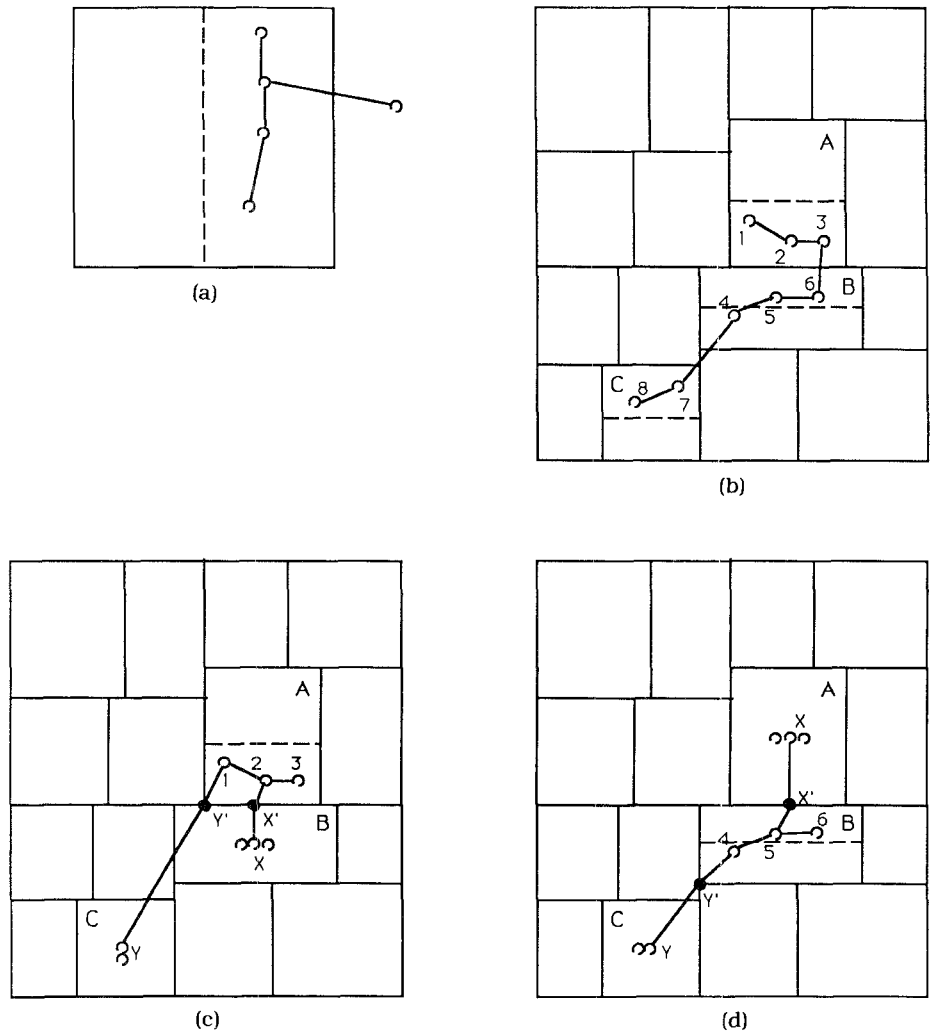


Figure 19. Terminal propagation. ○, real module; ●, dummy module.

sufficient to consider only the internal nets of the circuit, which may intersect the cut line. Nets connecting external terminals or other modules in another partition (at a higher level) must also be considered. Dunlop and Kernighan [1985] do this by a method called *terminal propagation*. Figure 19 illustrates the need for terminal propagation. Figure 19a shows the first division of the entire circuit into two sections. If a module is connected to an external terminal on the right side of the chip, it should be preferentially assigned to the right side of the chip, and vice versa. If this constraint was not considered, then each half of the circuit could have been assigned to either side of the chip. Figure 19b shows the result after several levels of partitioning. A particular net has cells connected to it in sections A, B, and C as shown. When these sections are partitioned further, it would be preferable to place these cells in the bottom half of A but in the top half of C. The assignment in B does not affect the wire length. Dunlop and Kernighan [1985] implement terminal propagation as follows.

Consider the situation when A is being partitioned vertically and the net

connecting cells 1, 2, and 3 in A also connects other cells in B and C. Such cells in other partitions are assumed to be at the center of their partition areas (points X and Y in Figure 19c) and are replaced by dummy cells at the nearest points on the boundary of A (e.g., at X', Y'). Now, during partitioning, the net-cut would be minimized if the cells 1, 2, and 3 are placed in the bottom half of A. A similar process for B does not yield any preference (Figure 19d), as predicted above.

To do terminal propagation, the partitioning has to be done breadth first. There is no point in partitioning one group to finer and finer levels without partitioning the other groups, since in that case no information would be available about which group a module should preferentially be assigned to.

The algorithm is in production use as part of an automated design system. The algorithm has been tested on a chip with 412 cells and 453 nets. It yields areas within 10–20% and track densities within 3% of careful hand layouts. CPU time of the order of 1 h on a VAX 11/780 has been reported. The CPU time can be significantly improved using the Fiduccia-Mattheyses [1982] linear-time partitioning heuristics.

3.3. Quadrisection

Suaris and Kedem [1987] have suggested the use of quadrisection instead of bipartitioning to divide the chip vertically and horizontally in a single partitioning step (Figure 20a), resulting in a truly two-dimensional placement procedure, rather than adapting a basically one-dimensional partitioning procedure to solve the two-dimensional placement problem. The quadrisection algorithm used is an extension of the Kernighan–Lin [1970] and Fiduccia–Mattheyses [1982] heuristics.

Unlike the Kernighan–Lin algorithm described above, a module in one quadrant can be interchanged with modules in any of the other three quadrants. This gives 12 gain tables, each corresponding to a pair of quadrants. At each step, the pairwise interchange giving the highest gain is selected.

The cost function is computed as follows. Let the cells connected to net n and placed in quadrant K be denoted by $\alpha_K(n)$. Then the *cell-distribution vector* for the net n is

$$\alpha(n) = \langle \alpha_1(n), \alpha_2(n), \alpha_3(n), \alpha_4(n) \rangle .$$

Associated with each net is a *resident flag vector*,

$$\beta(n) = \langle \beta_1(n), \beta_2(n), \beta_3(n), \beta_4(n) \rangle ,$$

such that

$$\beta_K(n) = \begin{cases} 1 & \text{if } \alpha_K(n) > 0 \\ 0 & \text{otherwise.} \end{cases}$$

Thus, the Kth component of $\beta(n)$ indicates whether any cells connected to net n are in quadrant K.

The cost function is defined as

$$W = \sum_{n \in N} w_n(\beta(n)),$$

where $w_n(\beta(n))$ is the cost of net n. If two or more components of $\beta(n)$ are nonzero, then there are cells connected to that net in the corresponding quadrants, and the net is being cut. The weights w_h and w_v are associated with horizontal and vertical net-cuts, respectively. The relative values of these weights indicate the preference in wiring direction. According to Suaris and Kedem [1987], in double- and triple-metal technology, where almost the entire space over the cells can be used for wiring, we would prefer vertical (over the cell) wiring. This would conserve channel space, which would otherwise be needed for horizontal wiring spans. Hence, in such technologies, w_v is usually set much less than w_h.

If all modules connected to a net are in horizontally adjacent quadrants, then the cost $w_n(\beta(n)) = w_h$. Similarly, if they are in vertically adjacent quadrants, then $w_n(\beta(n)) = w_v$. If the modules are in diagonally opposite quadrants or if they are distributed over any three quadrants, then $w_n(\beta(n)) = w_h + w_v$. If the modules connected to a net n are distributed over

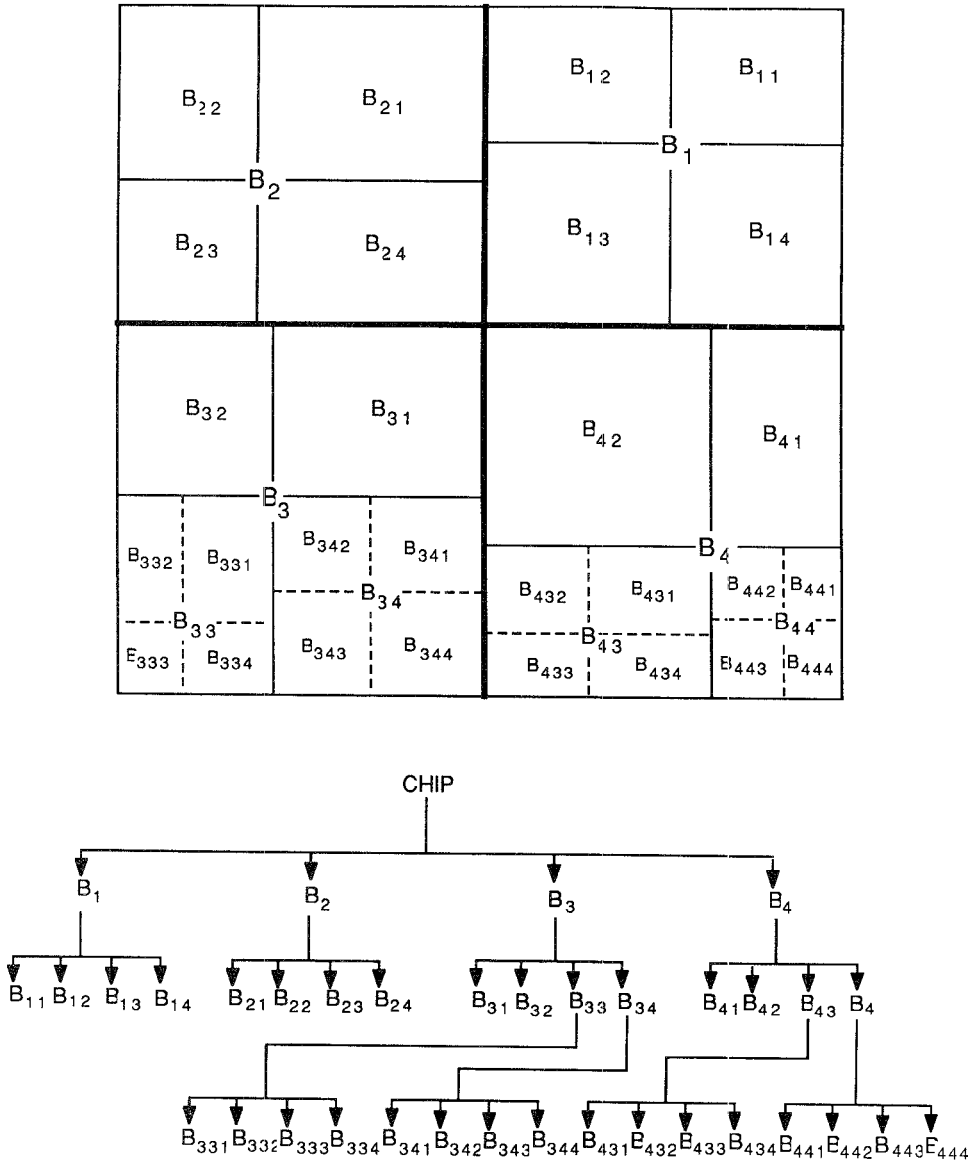


Figure 20a. Quadrisection.

all four quadrants, there are two possible interconnection patterns—one with one horizontal and two vertical cuts and the other with one vertical and two horizontal cuts. If $w_v < w_h$ as described above, we choose the first pattern and $w_n(\beta(n)) = 2w_v + w_h$.

If the cost function $w_n(\beta(n))$ is such that it can be computed from $\beta(n)$ in linear time, it can be proved that the quadrisection algorithm also runs in linear time. The rest of the partitioning algorithm is the same as in Fiduccia and Mattheyses [1982] and Kernighan and Lin [1970].

The terminal propagation method introduced by Dunlop and Kernighan [1985] has been extended for quadrisection as shown in Figure 20b. The figure shows regions 5 and 6 about to be partitioned

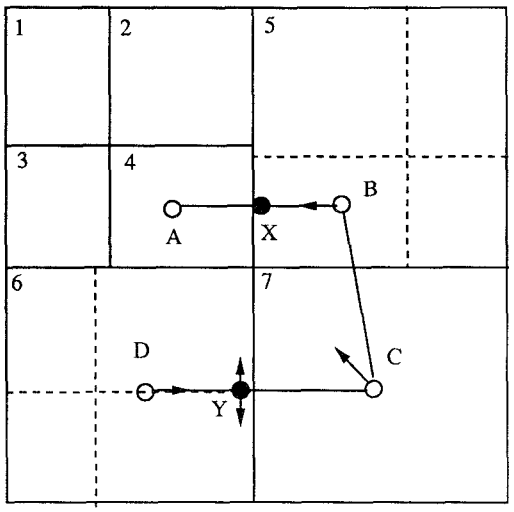


Figure 20b. Terminal propagation in quadrisection.

along the dotted lines and cells B and D in these regions connected to cells A and C in other regions. In this example, it would be beneficial to assign B to the lower left quadrant of region 5, as shown, and D to the upper or lower right quadrant of region 6 (since the exact position of C has not been determined yet). Terminal propagation is done by inserting two kinds of dummy cells—fixed and partially fixed at appropriate locations. Thus, in region 5, the influence of A is represented by a dummy cell X fixed in the lower left quadrant. The dummy cell will bias B into the same quadrant in order to reduce net-cut. In region 6, the cell C is represented by a partially fixed dummy cell Y, which is restricted to the upper and lower right quadrants. This will bias D into one of these quadrants.

Global routing information is also used to improve the efficiency of terminal propagation. For example, in Figure 20b, cells connected to the same net are located in all four quadrants and are to be connected as shown. Here, A and B should influence each other's position through terminal propagation, and so should B and C. Since there is no direct connection between A and C, however, there is no need for propagating them. After each partitioning step, the cells in different quadrants are connected in a pattern that gives the minimum cost, as discussed above. As the partitioning proceeds, these connection patterns give a global routing tree for each net. In the terminal propagation phase, only those modules that are directly connected to each other are propagated. The arrows show the effect of terminal propagation. For example, cell C will be biased toward the upper left quadrant when region 7 is quadrisectioned.

The algorithm has been implemented as part of the VPNR place and route package. Preliminary experiments show that this algorithm compares favorably with TimberWolf 3.2. For various standard cell circuits, this algorithm yielded an area within $\pm 5\%$ of the area yielded by TimberWolf. This algorithm, however, achieved this layout quality 50–200 times faster than TimberWolf. Run times reported are of the order of 1.4 min for a 304-cell circuit and 1 h for a 2907-cell circuit on a VAX 8600.

3.4 Other Techniques

Many variations of the min-cut placement algorithm have been suggested. Lauther [1979] applies this method for the placement of macro cells and uses

repeated partitioning to generate mutually dual placement graphs. His implementation also includes an improvement phase in which module rotation, mirroring, and other compression techniques are used.

Corrigan [1979] has developed another implementation of placement based on partitioning. Wipfler et al. [1982] discusses a combined force and cut algorithm for placement. Shiraishi and Herose [1980] have developed a min-cut based algorithm for master slice layout.

3.5. Analysis

The strength of min-cut algorithms is that they partition the problem into semi-independent subproblems. The concept of minimum net-cut implies a minimum amount of interaction between the parts that are placed independently. Dividing the problem into small parts brings about a drastic reduction in the factorial search space.

Partitioning can be thought of as a *successive approximation method* for placement. At each level of partitioning, the modules are localized in the region of the chip in which they ought to be finally located, but their exact position is not fixed. As the circuit is further partitioned and the smaller groups of modules are assigned to smaller chip areas, we get a better approximation of their final coordinates. This algorithm is less susceptible to local minima because the coordinates of all modules are being approximated simultaneously, with mutual benefit.

The problem with this technique is that partitioning is itself an NP complete problem and, therefore, is computationally intensive. This method is used for placement because heuristics developed so far for partitioning are much better in terms of speed and performance than those for placement. Note, however, that obtaining an optimal partitioning does not guarantee an optimal placement, although it would be close.

Overall, the results obtained from placement by partitioning algorithms are second only to simulated annealing. Beside, these algorithms take much less CPU time.

4. NUMERICAL OPTIMIZATION TECHNIQUES

Grouped together in this section are some computationally intensive deterministic techniques based on equation solving and eigenvalue calculations or on numerical optimization, such as the Simplex method. So far these techniques have mainly been used for macro blocks. The main problem encountered in using these techniques is that the placement problem is nonlinear. Two different approaches are used to overcome this obstacle. One method is to approximate the problem by a linear problem, then use linear programming. The other method is to use the various nonlinear programming methods [Walsh 1975]. Examples of both methods are given in the following sections.

4.1 Eigenvalue Method

Quadratic Assignment Problem [Gilmore 1962]. Given a cost matrix c_{ij} representing the connection cost of elements i and j and a distance matrix d_{kl} representing the distance between *locations* k and l, find a permutation function, p, that maps elements $i, j, \ldots$ to locations $k = p(i)$, $l = p(j), \ldots$ such that the sum

$$\Phi = \sum_{i,j} c_{ij} d_{p(i)p(j)}$$

is minimized. Consider the placement problem, where c_{ij} is the connectivity between cell i and cell j and d_{kl} is the distance between slot k and slot l. The permutation function p maps each cell to a slot. The wire length is given by the product of the connectivity and the distance between the slots to which the cells have been mapped. Thus, Φ gives the total wire length for the circuit, which is to be minimized. Hall [1970] has formulated the cell placement problem as a quadratic assignment problem and devised a novel method to solve it by using eigenvalues.

Let C be the connection matrix. Let c_i be the sum of all elements in the ith row of C. Define a diagonal matrix D such that

$$d_{ij} = \begin{cases} 0, & i \neq j, \\ c_i, & i = j. \end{cases}$$

The matrix B is defined as

$$B = D - C.$$

Further, let $X^T = [x_1, x_2, \ldots, x_n]$ and $Y^T = [y_1, y_2, \ldots, y_n]$ be row vectors representing the x- and y-coordinates of the desired solution. Then it can be proved [Hall 1970] that

$$\Phi(X, Y) = X^T B X + Y^T B Y.$$

Thus the problem is reduced to minimizing $\Phi(X, Y)$ subject to the quadratic constraints

$$X^T X = 1 \quad \text{and} \quad Y^T Y = 1.$$

These constraints are required to avoid the trivial solution $x_i = 0$ for all i. The minimization is done by introducing the Lagrange Multipliers α and β and forming the lagrangian

$$L = X^T B X + Y^T B Y - \alpha(X^T X - 1) - \beta(Y^T Y - 1).$$

Equating the first partial derivatives of L with respect to X and Y to zero, we get

$$2BX - 2\alpha X = 0 \qquad 2BY - 2\beta Y = 0$$

or

$$(B - \alpha I)X = 0 \qquad (B - \beta I)Y = 0.$$

These equations yield a nontrivial solution if and only if α and β are eigenvalues of the matrix B and X and Y are the corresponding eigenvectors. Premultiplying these equations by X^T and Y^T, respectively, and imposing the constraints $X^T X = 1$ and $Y^T Y = 1$, we get

$$\Phi(X, Y) = X^T B X + Y^T B Y = \alpha + \beta.$$

Thus, in order to minimize the value of the objective function Φ, we must choose the smallest eigenvalues as a solution for α and β. The corresponding eigenvectors X and Y will give the x- and y-coordinates of all the modules. If $0 = \lambda_1 < \lambda_2 < \lambda_3 < \cdots < \lambda_m$ are the distinct eigenvalues of B, then taking $\alpha = \beta = \lambda_1$ will give the minimum value $\Phi = 0$, x_i will be proportional to y_i, all x_i will be equal, and all y_i will be equal. If it is desired that X not be proportional to Y (i.e., we require a two-dimensional solution with all modules not placed along a straight line), we must select different eigenvalues for α and β. Further, if it is desired that not all x_i or all y_i be equal, we should ignore $\lambda_1 = 0$. Thus, a near optimal nontrivial solution is $\alpha = \lambda_2$, $\beta = \lambda_3$. The components of the eigenvector associated with the second-smallest eigenvalue give the x-coordinates of all the modules, and the components of the eigenvector associated with the third-smallest eigenvalue give the y-coordinates of all modules.

4.1.1 Example

An example is given in Figure 21. The netlist for the problem is

$$N_1 = \{1,3\}; \quad N_2 = \{1,4\}; \quad N_3 = \{2,4\};$$
$$N_4 = \{2,3\}; \quad N_5 = \{2,3\}.$$

The C, D, and B matrixes are

$$C = \begin{bmatrix} 0 & 0 & 1 & 1 \\ 0 & 0 & 2 & 1 \\ 1 & 2 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{bmatrix}$$

$$D = \begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 2 \end{bmatrix}$$

$$B = \begin{bmatrix} 2 & 0 & -1 & -1 \\ 0 & 3 & -2 & -1 \\ -1 & -2 & 3 & 0 \\ -1 & -1 & 0 & 2 \end{bmatrix}$$

The eigenvalues of B are 0, 2, 2.586, and 5.414. The eigenvectors corresponding to the eigenvalues 2 and 2.586 are [1 −1 −1 1] and [1 −0.414 0.414 −1] (Table 3). These eigenvectors give the x- and y-coordinates, respectively, for all four modules.

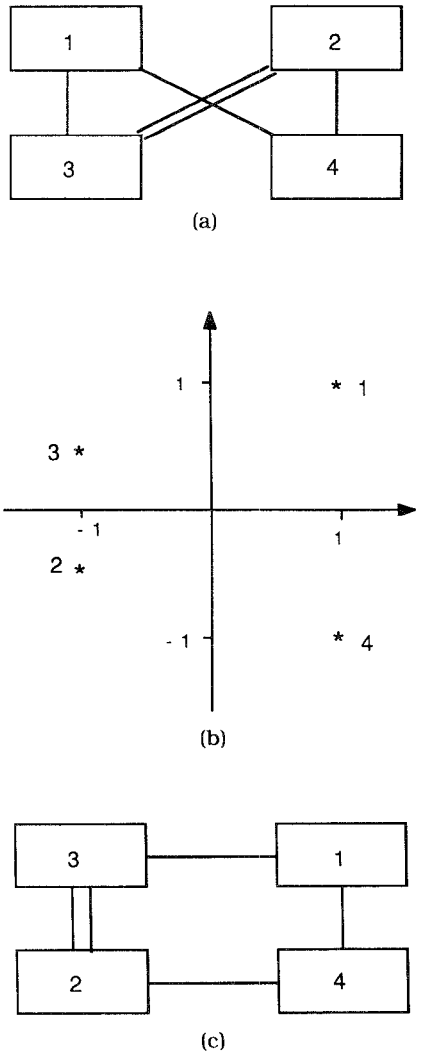


Figure 21. Placement by the eigenvalue method: An example. (a) Example circuit; (b) placement in the euclidean plane determined by the eigenvectors; (c) assignment to regularly spaced slots.

Table 3. Eigenvectors of Matrix B, Giving the Solution in Euclidean Space

Module	x-Coordinates	y-Coordinates
1	1	1
2	−1	−0.414
3	−1	0.414
4	1	−1
Eigenvalues	2	2.586

4.1.2 Analysis

This is an $O(n^2)$ algorithm. A weakness of the algorithm is that it does not take module size, shape, and routing channel width into account. It assumes that the modules are zero-area points. Therefore, it does not correspond very well to the module placement problem, where the modules must be placed at grid points or in rows. After this algorithm has determined the placement that minimizes the total wire length, mapping the modules form this placement to grid points can be very difficult for large circuits, with many ties requiring arbitrary decisions. The wire length is often increased significantly while converting the result of this algorithm to a legal placement.

4.2 Resistive Network Optimization

Cheng and Kuh [1984] have devised a novel technique for placement. They have transformed the placement problem into the problem of minimizing the power dissipation in a resistive network. The objective function (squared euclidean wire length) is written in matrix form, which yields a representation similar to the matrix representation of resistive networks. The placement problem is solved by manipulating the corresponding network to minimize power dissipation using sparse matrix techniques.

4.2.1 Objective Function and Analogy to Resistive Networks

The wire length is taken as the square of the euclidean distance between connected modules:

$$\Phi(X, Y) = \frac{1}{2} \sum_{i,j=1}^{n} c_{ij}\left[(x_i - x_j)^2 + (y_i - y_j)^2\right],$$

where c_{ij} is the connectivity between modules i and j. This can be written as

$$\Phi(X, Y) = X^T B X + Y^T B Y,$$

where $B = D - C$ as defined in the eigenvalue method described in Section 4.1. If this equation is compared to

Table 4. Analogy Between the Placement Problem and Power Dissipation in a Resistive Network

Resistive network	Placement problem
Power, P	Wire length, Φ
Nodes	Modules
Active voltage sources, v_2	Fixed module coordinates, x_2
Passive node voltages, v_1	Movable module coordinates, x_1
Admittance, Y	Connectivity, B

the equation for power dissipation in a resistive network,

$$P = V^T Y_n V,$$

we find that B is of the same form as the indefinite admittance matrix Y_n of an n-terminal linear passive resistive network. The coordinate x_i is analogous to the voltage at node i. The connectivity c_{ij} is analogous to the mutual conductance between nodes i and j, and d_{ii} is analogous to the self-admittance at node i. If the given netlist contains some fixed modules, such as pads located at the chip boundary, then that will be equivalent to having a fixed voltage at the corresponding nodes in the resistive network. Thus, fixed modules are equivalent to voltage sources. This analogy is summarized in Table 4. In a resistive network (Figure 22), the current always distributes itself so as to minimize the power dissipation. Hence the problem reduces to solving the network equations for current. This current will then give the optimal power dissipation and hence optimal placement. If there are no pads or other fixed modules, that case would be analogous to a passive resistive network with no voltage sources. All currents would be zero, which would yield a placement with all modules placed at the center of the chip. Hence fixed modules, preferably at the periphery, are required to spread the other modules out. Even then, modules are mostly clustered near the center. This algorithm uses scaling and relaxation as described below to spread them out over the entire chip.

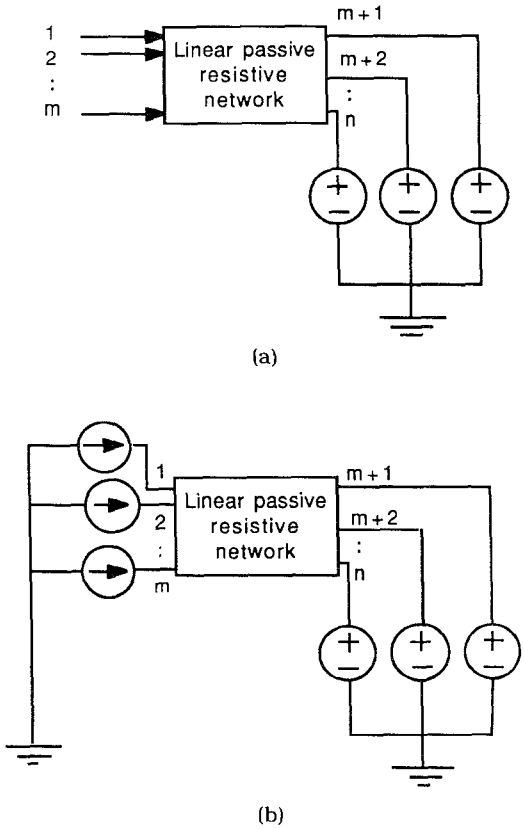


Figure 22. Cell placement by resistive network optimization. (a) n-terminal linear resistive network with m terminals floating and n-m terminals connected to voltage sources; (b) resistive network with linear constraints.

4.2.2 Slot Constraints

Slot constraints are required to guarantee module placement at grid points or *legal values*. A permutation vector p is defined such that the ith component p_i is the ith legal value or *slot* available for a module to occupy:

$$p = [p_1, p_2, \ldots, p_m]^T.$$

Let the position vector be

$$v_1 = [x_1, x_2, \ldots, x_m]^T.$$

The placement problem then consists of mapping each module to one slot; that is, associating each x_i with a p_j. The following slot constraints are necessary to yield a legal solution:

$$\sum_{i=1}^{m} x_i = \sum_{i=1}^{m} p_i$$
$$\sum_{i=1}^{m} x_i^2 = \sum_{i=1}^{m} p_i^2$$
$$\vdots$$
$$\sum_{i=1}^{m} x_i^m = \sum_{i=1}^{m} p_i^m.$$

The proof is given in Cheng and Kuh [1984]. As a simple example, consider a four module placement problem, with four given slot coordinates, $p_1, \ldots, p_4$. Then the assignment

$$x_1 = p_2; \quad x_2 = p_4;$$
$$x_3 = p_3; \quad x_4 = p_1$$

is a legal placement. It is easy to see that all the above constraints are satisfied. If two modules overlap, however,

$$x_1 = x_2 = p_2; \quad x_3 = p_3; \quad x_4 = p_1,$$

the above constraints will not be satisfied. Using all of the above constraints in the optimization process is not easy computationally. If we use only the first few constraints, we will get a solution that satisfies the corresponding properties, but the modules will not be located at the exact slot locations. For example, the first constraint helps align the center of gravity of the modules with that of the slots. Hence, using only this constraint will cause the resulting placement to be centered in the chip area.

4.2.3 Procedure

The overview of the placement procedure is as follows. First, the given circuit is mapped to a resistive network, where the fixed modules and pads are represented as fixed voltage sources. The power dissipation in the network is minimized, using only the first slot constraint. This causes all the modules to cluster around the center of the chip. The next step is scaling, in which the second slot constraint is used to spread the modules. Then repeated partitioning and relaxation are performed. This process aligns the modules with the slot locations.

(1) *Optimization.* The power dissipation in the network is optimized using the linear slot constraint. The optimization is done by applying the Kuhn-Tucker formula,

$$v_1 = y_{11}^{-1}\left[-y_{12}v_2 + i_1\right],$$

where

$$i_1 = \frac{d + I^T y_{11}^{-1} y_{12} v_2}{I^T y_{11}^{-1} I} I.$$

The goal of the optimization method is to reduce the euclidean wire length; that goal is best achieved by clustering all the modules close to each other. The use of the first constraint only centers the module placement in the chip area. If there are no fixed coordinates around the periphery of the chip, the optimization step will yield a trivial solution with all modules located at the center. With some modules at the periphery, a minimum wire length solution like the one shown in Figure 23a is obtained (for the netlist of Figure 4a).

(2) *Scaling.* In order to spread out the modules, the higher order slot constraints are required. In the second step, Cheng and Kuh [1984] repeat the optimization procedure using the second (parabolic) constraint. This will increase the power dissipation compared to the optimal but impractical solution of the previous step. The objective now is to find a configuration that results in a minimum increase in power dissipation. Using this objective, Cheng and Kuh [1984]

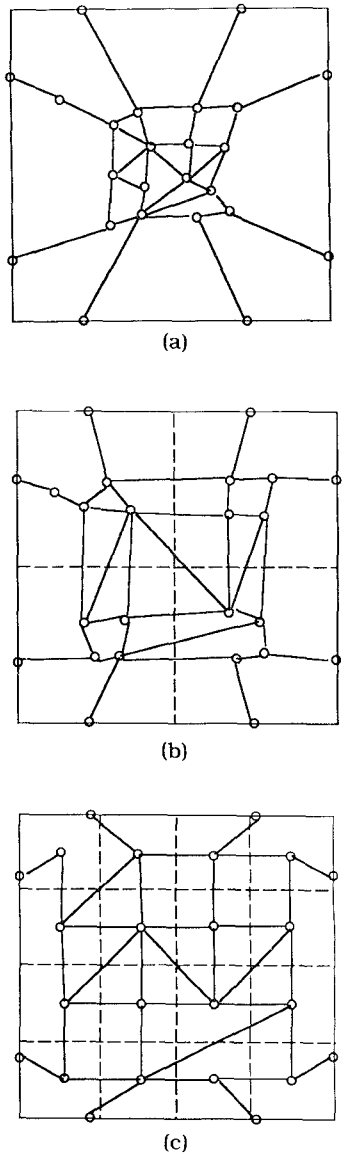


Figure 23. The partitioning step in resistive network optimization.

have derived the following equations, which give module coordinates that are more spread out:

$$x_{ni} = \frac{x_{oi} - c_0}{a_0} a_n + c_n,$$

where x_{oi} denote the solution after optimization, x_{ni} denote the new solution after scaling,

$$c_n = \frac{1}{k} \sum_{i=1}^{k} p_i$$

$$a_n = \left[\frac{1}{k} \sum_{i=1}^{k} (p_i - c_n)^2 \right]^{1/2}$$

$$c_0 = \frac{1}{k} \sum_{i=1}^{k} x_{oi}$$

$$a_0 = \left[\frac{1}{k} \sum_{i=1}^{k} (x_{oi} - c_0)^2 \right]^{1/2}.$$

(3) *Relaxation.* In this part of the algorithm, optimization and scaling are repeatedly done on subregions of size β specified by the user. First, optimization and scaling are done on one end region of size β, then on the other end region, and finally on the middle region. While doing optimization and scaling on one subregion, the rest of the modules are assumed to be fixed. By this process, the module positions are iteratively fine tuned.

(4) *Partitioning and assignment.* After the above steps, the modules are still not located exactly at the given slot locations. The next step is iterative partitioning into smaller and smaller regions. At each step, optimization and scaling are performed on the subregions according to the above equations. Every time, the linear slot constraint aligns the center of gravity of the group of modules with the center of the region in which it is being placed. At the last level of partitioning, when each section consists of only one module, the module is aligned to the center of the slot. This process is illustrated in Figures 23b and c.

4.2.4 Complexity and Results

Since linear network computations are required and sparse matrix techniques are used, the computation complexity is $O(m^{1.4})\log_2 m$, where m is the number of movable modules.

The algorithm has been tested on the 34 module example given by Steinberg [1961], and Hall [1970] and its performance compared against Steinberg's assignment algorithm and Hall's eigenvalue method. The wire length was 10% less compared to the eigenvalue method and 30% less compared to the Steinberg algorithm. A run time of 13.1 s was reported on a VAX 11/780. The performance was also compared to the algorithms of Stevens [1972] and Quinn and Breuer [1979] for a 136 module problem. The improvement in wire length was 9.5% over Stevens and 21% over Quinn and Breuer, and the CPU time was 104.2 s.

4.3 PROUD: Placement by Block Gauss-Seidel Optimization

Tsay et al. [1988] recently proposed an improved algorithm based on the resistive network analogy. The method consists of repeated solution of sparse linear equations. The slot constraints described above are bypassed, and the partitioning scheme is simplified. Block Gauss-Seidel (BGS) iteration is used to resolve the placement interactions between the blocks. The algorithm proceeds in two phases. First, global placement is done by the Successive Over Relaxation Method (SOR). This results in an optimal solution. The modules, however, are considered as zero-area points and are not confined to the grid points. Then, module shape and area are taken into consideration, and the chip is partitioned alternately in the vertical and horizontal direction. At each step BGS iteration is performed on each subregion in order to remove module overlap and successively approximate the module positions with the grid points. This process is repeated until each subregion consists of only one module.

4.3.1 Global Placement

First, the equations given in Section 4.4.2 are solved using SOR, which is a generalization of the BGS method. The method is as follows. To solve the equation

$$\mathbf{A}\mathbf{x}_1 = \mathbf{b},$$

$$\mathbf{A} = \mathbf{\Lambda}(\mathbf{L} + \mathbf{I} + \mathbf{U}),$$

where $\mathbf{\Lambda}$ is a diagonal positive definite matrix and $\mathbf{L}$ and $\mathbf{U}$ are lower and upper triangular matrices, respectively. The vector $\mathbf{x}_1$ is solved iteratively by the recursive formula

$$\mathbf{x}_1(k+1) = \mathbf{M}\mathbf{x}_1(k) + \mathbf{a},$$

where

$$\mathbf{M} = (\mathbf{I} + w\mathbf{L})^{-1}[(1-w)\mathbf{I} - w\mathbf{U}]$$

and

$$\mathbf{a} = w(\mathbf{I} + w\mathbf{L})^{-1}\mathbf{\Lambda}^{-1}\mathbf{b}.$$

The parameter w is in the range 0 to 2. With $w = 1$, the SOR method is reduced to the BGS method.

This method gives the global optimum solution because, in the absence of slot constraints, the objective function (the euclidean wire length) is convex and has a unique global minimum, which can be determined by solving the matrix equations.

4.3.2 Partitioning and BGS Iteration

The object of the partitioning and BGS iteration step is to ensure that for each subregion the total area of the modules placed on one side of the center line is equal to the total area of the modules placed on the other side of the center line. The partitioning is done as follows. Each cut is placed so that the total area of the modules on either side of the cut (as given by the global placement) is equal. If the cut line coincides with the center of the layout area, then the partition process is continued to the next hierarchy level; otherwise, the following method is used to align the cut with the center of the subregion.

Let the cut be to the right of the center. Then all modules to the right of the center line are projected to the center line, only those modules that lie between the cut line and the center line are considered as movable, and the global placement phase is repeated in the left half

plane. Then the modules in the left half plane are projected on the center line as fixed modules, and the global placement problem is solved for the right half plane. This procedure will align the cut line with the center line of the subregion being divided. The partitioning is repeated alternately in the horizontal and vertical directions until each subregion contains only one module.

In order to explain the intuitive concepts behind this method, Tsay et al. [1988] gave an analogy with min-cut algorithms. This algorithm can be considered as a form of min-cut algorithm, which uses quadratic assignment instead of the Kernighan–Lin heuristics for partitioning. An optimal placement results in a min-cut partition at any cut line through it. Thus, we can repeatedly determine the optimum but irregular placement of point modules in the euclidean plane by solving the quadratic assignment problem and subdivide the plane to get a min-cut partition. If this process is repeated until each partition consists of only one module, we get a near-optimal placement with no overlaps, and the modules constrained to grid locations, just like in the min-cut algorithm. The quadratic assignment problem can be solved using powerful sparse matrix techniques.

4.3.3 Complexity and Results

The algorithm has been implemented in the Proud-2 placement system. It was tested on nine circuits consisting of 1000–26,000-modules. In all cases, the results were superior to those of TimberWolf 3.2 and comparable to those of TimberWolf 4.2. The time required to achieve these results was about 50 times less compared to TimberWolf 4.2. For example, a 26,000-module circuit required a run time of about 50 min on a VAX 8650. For a 1438 module example, Proud-2 required 50 s, TimberWolf 3.2 required 7200 s, and TimberWolf 4.2 required 3260 s. Compared to the wire length achieved by Proud-2, the results of TimberWolf 3.2 were 7.1% worse, and the results of TimberWolf 4.2 were 9.6% better.

4.4 ATLAS: Technique for Layout Using Analytic Shapes

Sha and Blank [1987] (and earlier Sha and Dutton [1985]) used the Penalty Function Method (PFM), a nonlinear numerical optimization method, for block placement. They devised a modified objective function for macroblocks, which allows computationally efficient rotation and mirroring. They also made an excellent comparison between simulated annealing and numerical techniques.

4.4.1 Objective Function

The objective function used to estimate the wire length is the same as that described in Sha and Dutton [1985], with modifications to accommodate block rotation and mirroring. The original objective function is as follows:

Let S_k be a net connected to m_k blocks, with centers at $C_1(x_1, y_1), \ldots, C_2(x_2, y_2), \ldots, C_{m_k}(x_{m_k}, y_{m_k})$ and the center of gravity of the net S_k be G_k $(\bar{x}_k, \bar{y}_k)$, where

$$\bar{x}_k = \frac{1}{m_k} \sum_{i=1}^{m_k} x_i \qquad \bar{y}_k = \frac{1}{m_k} \sum_{i=1}^{m_k} y_i.$$

The squared wire length of the net S_k (Figure 24a) is defined as

$$w_k = \sum_{i=1}^{m_k} \{(x_i - \bar{x}_i)^2 + (y_i - \bar{y}_i)^2\}.$$

If m_s is the total number of nets, the objective function is defined as

$$W = \sum_{j=1}^{m_s} w_k = \sum_{j=1}^{m_s} \sum_{i=1}^{m_k} \{(x_i - \bar{x}_i)^2 + (y_i - \bar{y}_i)^2\}.$$

In macro placement, block orientation is important besides block position because of two reasons. In order to fit the irregularly shaped blocks together while minimizing the wasted space, all possible

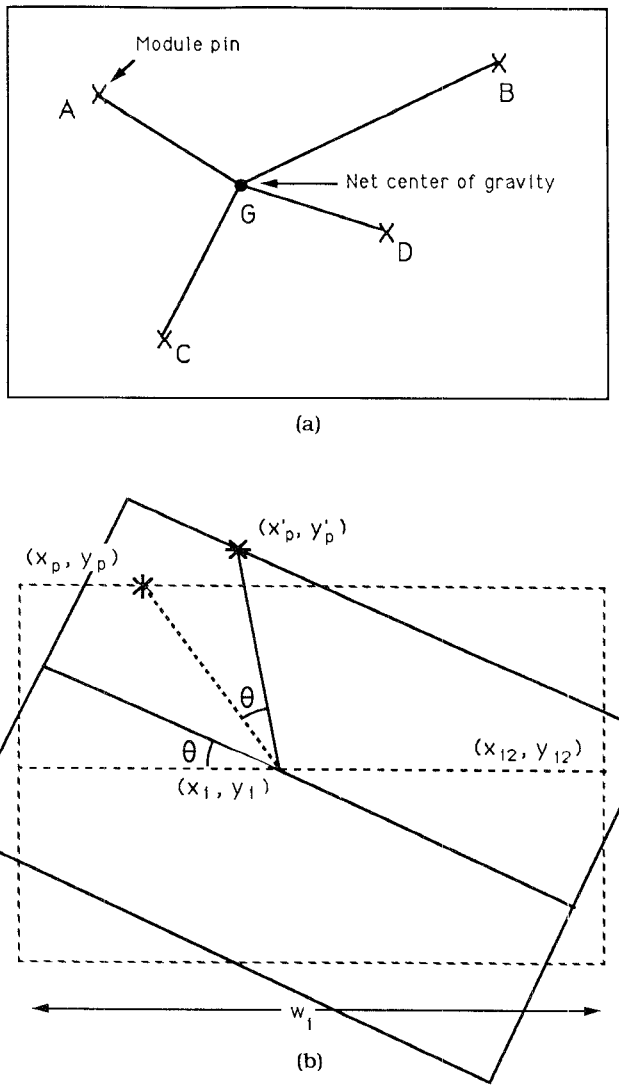


Figure 24. Squared euclidean wire length function, $w_k = AG^2 + BG^2 + CG^2 + DG^2$; (b) pin position after rotation in Atlas.

orientations should be allowed. Besides, rotation and mirroring have a significant effect on the wire length. If the pins on one side (left, say) of a large block are connected to other blocks placed on the other side (right), then the nets are forced to go around the block. Flipping the block over reduces the wire length.

Block Rotation. If a block is rotated through an angle θ, the pin coordinates (x'_p, y'_p) relative to the block center are given by

$$x'_p = x_p \cos\theta - y_p \sin\theta,$$

$$y'_p = y_p \cos\theta + x_p \sin\theta,$$

where x_p and y_p are the original coordinates relative to the block center (Figure 24b). Let the block axis be defined by the pair of coordinates (x_{i1}, y_{i1}) and (x_{i2}, y_{i2}) and the block center be denoted

by (x_i, y_i). Let

$$\Delta x_i = |x_{i1} - x_{i2}|,$$
$$\Delta y_i = |y_{i1} - y_{i2}|,$$
$$d_i = \sqrt{\Delta x_i^2 + \Delta y_i^2}$$
$$\alpha_1 = \frac{x_p}{d_i},$$
$$\alpha_2 = \frac{y_p}{d_i}.$$

Then, we get the absolute pin position after rotation:

$$x_{pi} = x_i + \alpha_1 \Delta x_i - \alpha_2 \Delta y_i,$$
$$y_{pi} = y_i + \alpha_1 \Delta y_i + \alpha_2 \Delta x_i.$$

The new parameters introduced in the objective function are constants x_p and y_p. Since no new variables are introduced, there is little increase in computation time.

Mirroring. The mirroring operation is realized by introducing an extra variable, u_i, such that $u_i = 1$ means normal orientation and $u_i = -1$ means mirrored orientation. The pin position is now given by

$$x_{pi} = x_i + \alpha_1 \Delta x_i - u_i \alpha_2 \Delta y_i,$$
$$y_{pi} = y_i + \alpha_1 \Delta y_i + u_i \alpha_2 \Delta x_i.$$

These pin coordinates are used along with the block coordinates in order to calculate the wire length more accurately. During optimization, u_i is treated as a continuous variable that can vary within the bounds $|u_i| \leq 1$. A constraint $(u_i - 1)(u_i + 1) = 0$ is imposed during the optimization process. This constraint makes u converge to either $+1$ or -1.

4.4.2 Constraint Conditions

In addition to the above objective function, some constraints are imposed during the solution process in order to ensure a legal placement. These constraints are only summarized here. For a detailed discussion, see Sha and Dutton [1985].

Let the block width be w_i and the block height be h_i, with $l_i = w_i - h_i$, and Δx_i, Δy_i, and d_i be as defined above. The constraint that prevents block overlap is given by

$$g_1(i, j) = r_i + r_j - d(i, j) \leq 0;$$

r_i, r_j are the block *radii* (half the block height), and $d(i, j)$ is the distance between the axes of the blocks i and j. For the derivation, see Sha and Dutton [1985].

Only two block orientations, vertical and horizontal, are allowed. To ensure this, the orientation constraint is used:

$$g_2(i) = \Delta x_i \Delta y_i / l_i = 0 \quad \text{for} \quad i = 1, 2, \ldots, m.$$

It is obvious that if the block is not vertical or horizontal, both Δx_i and Δy_i will be nonzero, and the constraint will not be satisfied.

The constraint for the desired block size is given by

$$g_3(i) = l_i - d_i = 0.$$

Let the desired chip aspect ratio be q, with the vertical and horizontal dimensions y_m and $x_m = qy_m$, respectively. Then, the boundary constraints are

$$g_{41l}(i) = r_i - x_{il} \leq 0,$$
$$g_{42l}(i) = r_i - y_{il} \leq 0,$$
$$g_{43l}(i) = x_{il} + r_i - x_m \leq 0,$$
$$g_{44l}(i) = y_{il} + r_i - y_m \leq 0,$$

for $l = 1, 2$; $i = 1, 2, \ldots, m$.

4.4.3 Penalty Function Method

The penalty function method consists of the following procedure:

(1) Select an increasing series c_k, typically

$$c_0 = 1; \qquad c_{k+1} = 10 c_k.$$

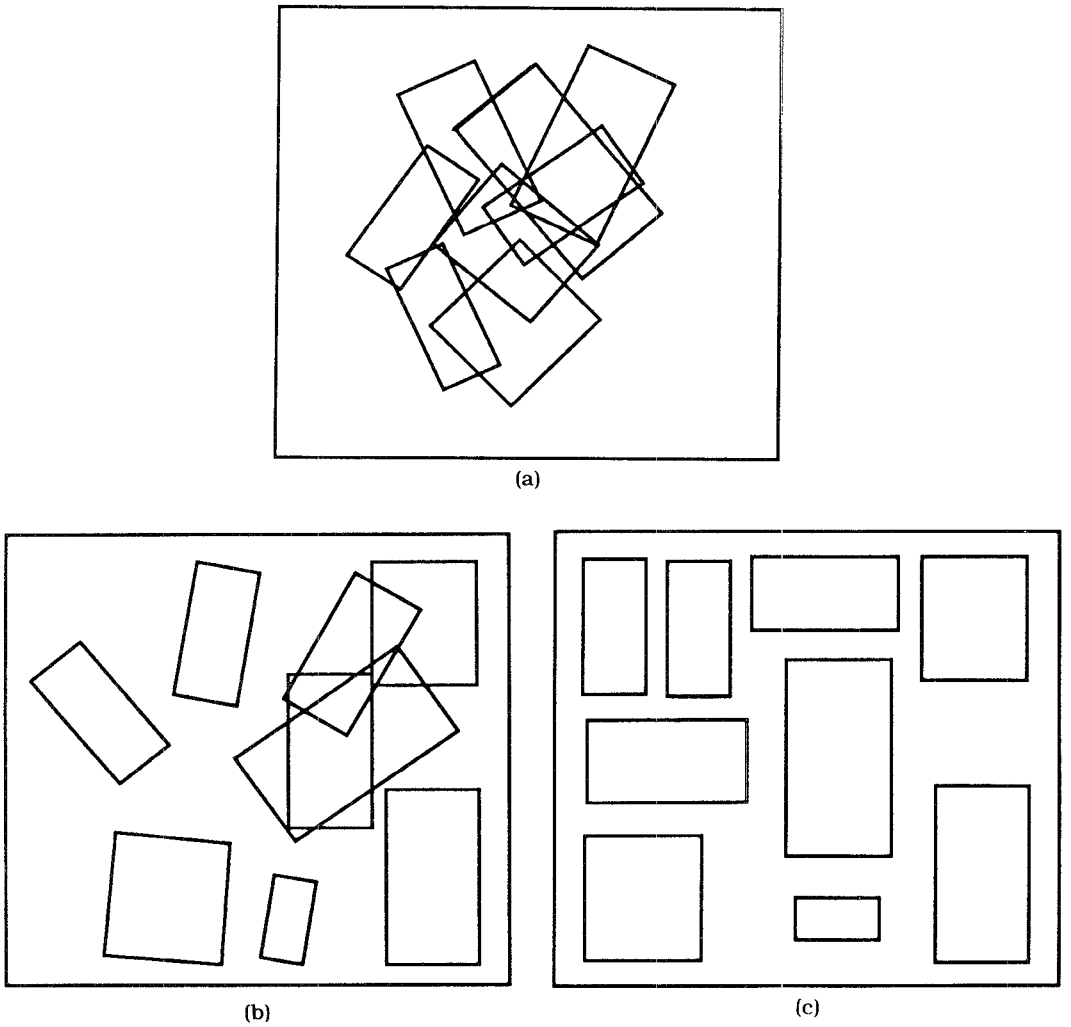


Figure 25. PFM optimization process: Intermediate results as c_k is increased.

(2) Construct a new unconstrained objective function $P(x, c_k)$ such that

$$P(x, y, c_k) = \textit{objective function} + c_k \sum \textit{constraints}.$$

(3) Use an unconstrained optimization technique such as Newton's method to minimize $P(x, y, c_k)$ for $k = 0, 1, 2, \ldots$, until the coordinates of the modules satisfy the constraints within the required accuracy.

Thus, in PFM during the first iteration, the constraints are deemphasized and a solution is obtained. Figure 25a shows the result of the first iteration, with $c_k = 1$. Then, in each iteration the weight of the constraints is increased and the objective function minimized. This causes the constraints to be satisfied more and more accurately, at the expense of an increase in the value of the objective function. An intermediate result is shown in Figure 25b. As c_k is increased, the modules attain the proper orientation and overlap is reduced. The process is terminated when the constraints are satisfied within the desired accuracy. Figure 25c shows the final result, with all modules

in either vertical or horizontal orientation and no overlap.

4.4.4 Comparison with Simulated Annealing

PFM uses numerical techniques, whereas simulated annealing uses a statistical approach. Although the techniques of nonlinear programming and simulated annealing are very different, some similarities exist. The parameter c_k in PFM behaves like the reciprocal of temperature $(1/T)$ in simulated annealing. In simulated annealing, moves are randomly generated, whereas in PFM moves are deterministic and are in the direction that minimizes the penalty function $P(x, y, c_k)$. The important feature of PFM is that all blocks move simultaneously, not one at a time as in simulated annealing.

PFM has been tested on two chips with 23 and 33 macro cells, and the results have been compared to those of TimberWolf and industrial placement. A 50% improvement over industrial placement and 23% improvement over TimberWolf were reported. The CPU time reported is of the order of 2 h on a VAX station II for a 33-block circuit.

4.5 Algorithm for Block Placement by Size Optimization

One example of linearization is provided by Mogaki et al. [1987]. They presented an algorithm for the placement of macro blocks to minimize chip area under constraints on block size, relative block position, and the width of the available interblock routing space. This algorithm iteratively determines the optimum block size and relative block placement in order to reduce the wasted space and minimize the total chip area. Channel widths are also considered during the optimization process. This is a quadratic integer programming problem, which has been reformulated as a linear programming problem and solved by the Simplex method. This algorithm is the extension of the work of Kozawa et al. [1984], and uses their Combined and Top Down Placement (CTOP) algorithm for the actual block placement. This is coupled with block resizing by linear programming. This algorithm is suitable only where block sizes are not yet fixed and macro blocks can be generated with a range of possible aspect ratios. Hence, choosing the block aspect ratios to fit together nicely in the placement, then generating the macro blocks results in a compact layout.

The first step is to generate an initial placement by the CTOP algorithm. This algorithm works by repeatedly combining two blocks to form a hyper-block until the entire chip consists of one hyper-block. At each step, blocks are paired so as to minimize the wasted space and maximize their connectivity to each other. Repeated combining of blocks generates a *combine* tree with the entire chip as the root node and the individual blocks as leaves. This tree is then traversed top down, such that for each hyper-block, a good placement is determined for its component blocks. This gives the relative placement of the blocks.

The relative placement is converted to a *Block Neighboring Graph* (BNG), as shown in Figure 26. Each block is represented as a node in the BNG, and each segment of a routing channel between two blocks is represented by an edge connecting the nodes. Formally,

$$\text{BNG} = G(V, E)$$

such that

$$V = \{v\} U \{L, R, B, T\};$$
$$E \subset V \times V \times \{X, Y\} \times \{\delta\},$$

where v represents a block, L, R, B, T are the simulated blocks corresponding to left, right, top, and bottom edges of the chip, X or Y represent whether the channel is vertical or horizontal, and $\delta > 0$ represents the minimum channel width.

The next step is the linear programming formulation for block size optimization. This consists of the objective function and the constraints.

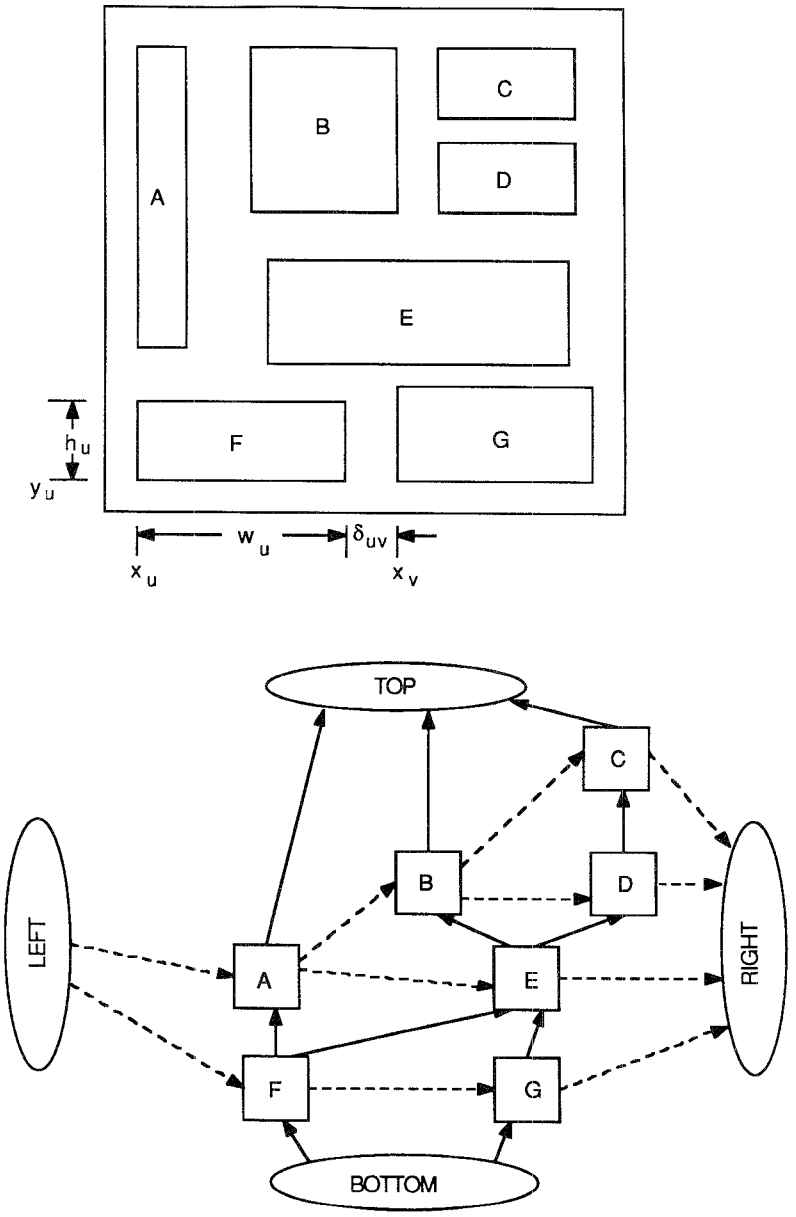


Figure 26. Example macro-block layout and its block neighboring graph. □, Block; ⬭, simulated block; →, vertical edge; ---> , horizontal edge.

4.5.1 Objective Function

The primary objective is to minimize the chip area. Wire length is considered indirectly through its effect on the chip area. There is a user-specified limit on the chip aspect ratio. Let the minimum and maximum desirable values of the chip aspect ratio be r^- and r^+:

$$r^- \leq \frac{y}{x} \leq r^+.$$

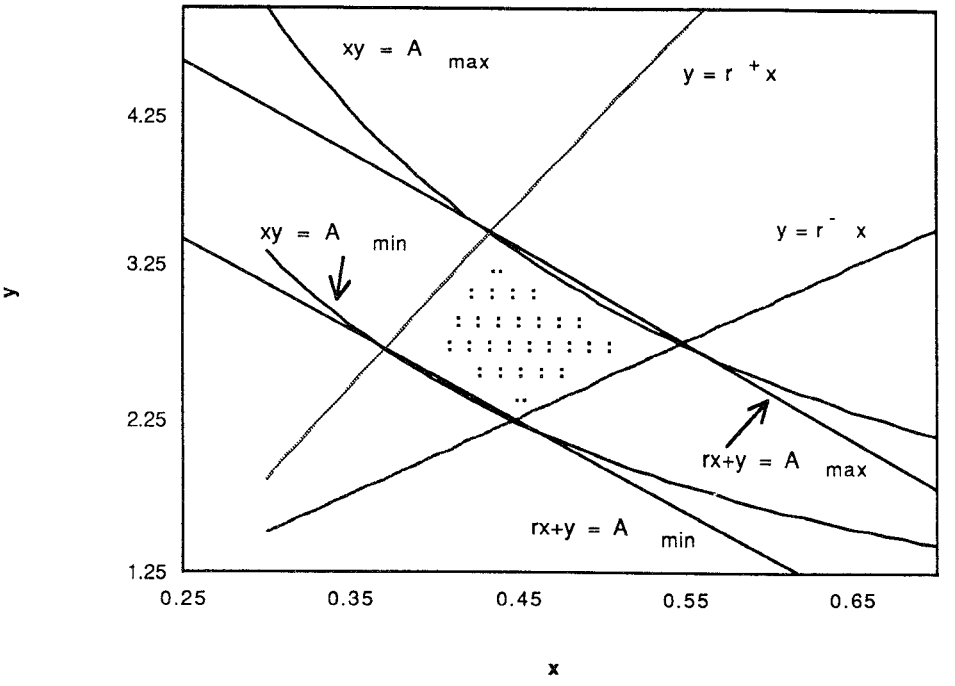


Figure 27. Linear programming: objective function.

If r^- and r^+ are sufficiently close, we get

$$r = \frac{y}{x} \approx \sqrt{r^- r^+}.$$

The chip area is given by

$$A = xy \approx \frac{1}{4r}(rx + y)^2.$$

Since the function $(rx + y)^2$ increases monotonically with $(rx + y)$ for $x, y > 0$, the chip area function can be replaced by $rx + y$, which is a linear function, because r is a constant. Thus, in order to minimize chip area xy, the linear programming problem is formulated to minimize $rx + y$. The effect of this linearization is shown in Figure 27. The shaded region represents the allowed values of chip width and height. This region is bounded by the maximum and minimum chip area constraints and the maximum and minimum aspect ratio constraints. Within a small aspect ratio range, the linearized area is a good approximation to the actual area.

4.5.2 Constraints

(1) *Block size constraint.* Each block has a given range of candidate sizes given by $(w_v(i), h_v(i))$, where $i = 1, 2, \ldots, N_v$. N_v is the number of possible sizes for block v. The object of the linear programming approach is to choose the aspect ratio that results in the minimum wastage of chip area. Let $\lambda_v(i)$ be the selection weight for the ith candidate block size such that

$$\sum_{i=1}^{N_v} \lambda_v(i) = 1, \qquad 0 \le \lambda_v(i) \le 1.$$

$\lambda_v(i)$ represents the probability distribution for selection of each candidate block size. The expected block width and height are therefore given as

$$w_v = \sum_{i=1}^{N_v} \lambda_v(i) w_v(i)$$

$$h_v = \sum_{i=1}^{N_v} \lambda_v(i) h_v(i).$$

These are the block size constraints.

(2) *Channel width constraints.* Let x_u, y_u, x_v, y_v be the coordinates of blocks u and v, respectively, and w_u and h_u be the width and height of block u. If block v is to the right of block u, with a channel of width δ_{uv} between them, we have

$$x_v - (x_u + w_u) \geq \delta_{uv}$$

for each edge

$$(u, v, X, \delta_{uv}) \in E,$$

as shown in Figure 26. Similarly, if block u is below block v, we get the corresponding constraint

$$y_v - (y_u + w_u) \geq \delta_{uv}$$

for each edge

$$(u, v, Y, \delta_{uv}) \in E.$$

These constraints make it possible to determine an appropriate block size when the channel widths are specified.

Thus, the objective of the linear programming formulation is to determine x_u, y_u, w_u, h_u for all blocks u so that the linearized area $rx + y$ is minimized, subject to the above constraints.

4.5.3 Procedure

The algorithm can be summarized as follows:

(1) Determine the relative block placement by the CTOP algorithm [Kozawa et al. 1984].
(2) Determine the absolute block placement by the following repetitive procedure:
  (2.1) Determine the channel width by global routing.
  (2.2) Optimize block size using the following optimization algorithm:
    (2.2.1) Generate the BNG from the relative block positions.
    (2.2.2) Eliminate redundant constraints and convert the rest into an LP condition matrix.
    (2.2.3) Solve the LP problem by the Simplex method.
    (2.2.4) Select the block size closest to the LP solution.
  (2.3) Go to 2.1

The algorithm has been tested on two chips with up to 40 macro blocks. Experimental results indicate that 6% saving of area can result over manual designs and 5–10% over other algorithms. This saving is achieved, however, at the cost of 10–12 times the computation time compared to other algorithms.

4.6 Other Work in This Field

Blanks [1985a, 1985b] has exploited the mathematical properties of the quadratic (sum of squares) distance metric to develop an extremely fast wire length evaluation scheme. He uses the eigenvalue method to determine the lower bound on the wire length in order to evaluate his iterative improvement procedures. He has also given a theoretical model to explain the observed deviation from optimality.

Markov et al. [1984] have used Bender's [1962] procedure for optimization. Blanks [1984] and Jarmon [1987] have used the least-squares technique. Akers [1981] has used linear assignment. He has given two versions—constructive placement and iterative improvement. Further work on the eigenvalue method has been done by Hanan and Kurtzberg [1972b]. Hillner et al. [1986] has proposed a dynamic programming approach for gate array placement (see also Karger and Malek [1984]). Herrigel and Fichtner [1989] have used the Penalty Function Method for macro placement. Kappen and de Bont [1990] have presented an improvement over Tsay et al.'s algorithm discussed in Section 4.3.

5. PLACEMENT BY THE GENETIC ALGORITHM

The genetic algorithm is a very powerful optimization algorithm, which works by emulating the natural process of evolution as a means of progressing toward

the optimum. Historically, it preceded simulated annealing [Holland 1975], but it has only recently been widely applied for solving problems in diverse fields, including VLSI placement [Grefenstette 1985, 1987]. The algorithm starts with an initial set of random configurations, called the *population*. Each individual in the population is a string of symbols, usually a binary bit string representing a solution to the optimization problem. During each iteration, called a *generation*, the individuals in the current population are *evaluated* using some measure of fitness. Based on this *fitness* value, individuals are selected from the population two at a time as parents. The fitter individuals have a higher probability of being selected. A number of genetic operators is applied to the parents to generate new individuals, called *offspring*, by combining the features of both parents. The three genetic operators commonly used are crossover, mutation, and inversion, which are derived by analogy from the biological process of evolution. These operators are described in detail below. The offspring are next evaluated, and a new generation is formed by selecting some of the parents and offspring, once again on the basis of their fitness, so as to keep the population size constant.

This section explains why genetic algorithms are so successful in complex optimization problems in terms of schemata and the effect of genetic operators on them. Informally, the symbols used in the solution strings are known as *genes*. They are the basic building blocks of a solution and represent the properties that make one solution different from the other. For example, in the cell placement problem, the ordered triples consisting of the cells and their assigned coordinates can be considered genes. A solution string, which is made up of genes, is called a *chromosome*. A *schema* is a set of genes that make up a partial solution. An example would be a subplacement, consisting of any number of such triples, with 'don't cares' for the rest of the cells. A schema with m defining elements and 'don't cares' in the rest of the $n - m$ positions (such as an m-cell subplacement in an n-cell placement problem) can be considered as an $(n - m)$-dimensional hyperplane in the solution space. All points on that hyperplane (i.e., all configurations that contain the given subplacement) are *instances* of the schema. Note here that the subplacement does not have to be physically contiguous, such as a rectangular patch of the chip area. For example, a good subplacement can consist of two densely connected cells in neighboring locations. Similarly, a good subplacement can also consist of a cell at the input end of the network and a cell at the output end that are currently placed at opposite ends of the chip. Both of these subplacements will contribute to the high peformance of the individual that inherits them. Thus, a schema is a *logical* rather than physical grouping of cell-coordinate triples that have a particular relative placement.

As mentioned above, the genetic operators create a new generation of configurations by combining the schemata (or subplacements) of parents selected from the current generation. Due to the stochastic selection process, the fitter parents, which are expected to contain some good subplacements, are likely to produce more offspring, and the bad parents, which contain some bad subplacements, are likely to produce less offspring. Thus, in the next generation, the number of good subplacements (or high-fitness schemata) tends to increase, and the number of bad subplacements (low-fitness schemata) tends to decrease. Thus, the fitness of the entire population improves. This is the basic mechanism of optimization by the genetic algorithm.

Each individual in the population is an instance of 2^n schemata, where n is the length of each individual string. (This is equivalent to saying that an n-cell placement contains 2^n subplacements of any size.) Thus, there is a very large number of schemata represented in a relatively small population. By trying out one new offspring, we get a rough estimate of the fitness of all of its schemata or subplace-

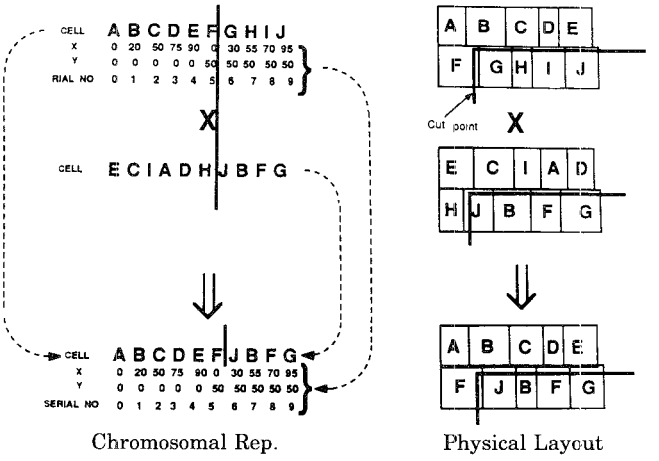


Figure 28. Traditional method of crossover. A segment of cells is taken from each parent. The coordinate array is taken from the first parent. With this method, cells B and F are repeated, and cells H and I are left out.

ments. Thus, with each new configuration examined, the number of each of its 2^n schemata present in the population is adjusted according to its fitness. This effect is termed the intrinsic parallelism of the genetic algorithm. As more configurations are tried out, the relative proportions of the various schemata in the population reflect their fitness more and more accurately. When a fitter schema is introduced in the population through one offspring, it is inherited by others in the succeeding generation; therefore its proportion in the population increases. It starts driving out the less fit schemata, and the average fitness of the population keeps improving.

The genetic operators and their significance can now be explained.

Crossover. Crossover is the main genetic operator. It operates on two individuals at a time and generates an offspring by combining schemata from both parents. A simple way to achieve crossover would be to choose a random cut point and generate the offspring by combining the segment of one parent to the left of the cut point with the segment of the other parent to the right of the cut point. This method works well with the bit string representation. Figure 28 gives an example of crossover. In some applications, where the symbols in the solution string cannot be repeated, this method is not applicable without modification. Placement is a typical problem domain where such conflicts can occur. For example, as shown in Figure 28, cells B and F are repeated, and cells H and I are left out. Thus, we need either a new crossover operator that works well for these problem domains or a method to resolve such conflicts without causing significant degradation in the efficiency of the search process. The performance of the genetic algorithm depends to a great extent on the performance of the crossover operator used. Various crossover operators that overcome these problems are described in the following sections.

When the algorithm has been running for some time, the individuals in the population are expected to be moderately good. Thus, when the schemata from two such individuals come together, the resulting offspring can be even better, in which case they are accepted into the population. Besides, the fitter parents have a higher probability of generating offspring. This process allows the algorithm to examine more configurations in a region of greater average fitness so the

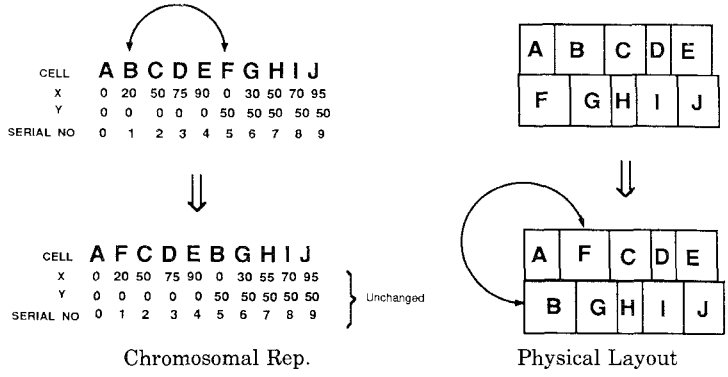


Figure 29. Mutation.

optimum may be determined and, at the same time, examine a few configurations in other regions of the configuration space so other areas of high average performance may be discovered.

The amount of crossover is controlled by the *crossover rate*, which is defined as the ratio of the number of offspring produced in each generation to the population size. The crossover rate determines the ratio of the number of searches in regions of high average fitness to the number of searches in other regions. A higher crossover rate allows exploration of more of the solution space and reduces the chances of settling for a false optimum; but if this rate is too high, it results in a wastage of computation time in exploring unpromising regions of the solution space.

Mutation. Mutation is a background operator, which is not directly responsible for producing new offspring. It produces incremental random changes in the offspring generated by crossover. The mechanism most commonly used is pairwise interchange as shown in Figure 29. This is not a mechanism for randomly examining new configurations as in other iterative improvement algorithms. In genetic algorithms, mutation serves the crucial role of replacing the genes lost from the population during the selection process so that they can be tried in a new context or of providing the genes that were not present in the initial population. In terms of the placement problem, a gene consisting of an ordered triple of a cell and its associated *ideal* coordinates may not be present in any of the individuals in the population. (That is, that particular cell may be associated with nonideal coordinates in all the individuals.) In that case, crossover alone will not help because it is only an inheritance mechanism for existing genes. The mutation operator generates new cell-coordinate triples. If the new triples perform well, the configurations containing them are retained, and these triples spread throughout the population.

The *mutation rate* is defined as the percentage of the total number of genes in the population, which are mutated in each generation. Thus, for an n-cell placement problem, with a population size N_p, the total number of genes is nN_p, and $nN_pR_m/2$ pairwise interchanges are performed for a mutation rate R_m. The mutation rate controls the rate at which new genes are introduced into the population for trial. If it is too low, many genes that would have been useful are never tried out. If it is too high, there will be too much random perturbation, the offspring will start losing their resemblance to the parents, and the algorithm will lose the ability to learn from the history of the search.

Inversion. The inversion operator takes a random segment in a solution string and inverts it end for end

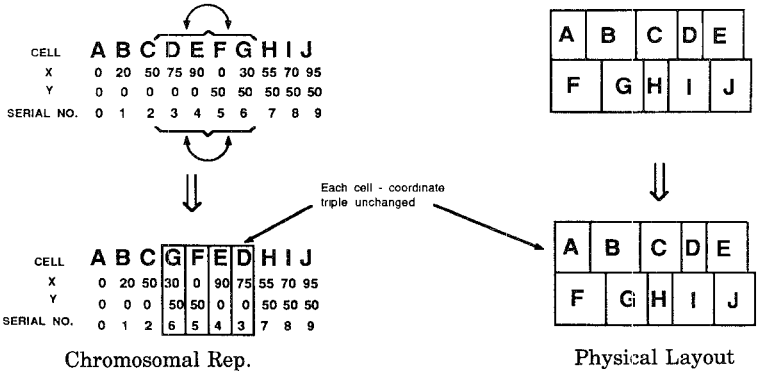


Figure 30. Inversion.

(Figure 30). This operation is performed in such a way that it does not modify the solution represented by the string; instead, it only modifies the *representation* of the solution. Thus, the symbols composing the string must have an interpretation independent of their position. This can be achieved by associating a serial number with each symbol in the string and interpreting the string with respect to these serial numbers instead of the array index. When a symbol is moved in the array, its serial number is moved with it, so the interpretation of the symbol remains unchanged. In the cell placement problem, the x- and y-coordinates stored with each cell perform this function. Thus, no matter where the cell-coordinate triple is located in the population array, it will have the same interpretation in terms of the physical layout.

The advantage of the inversion operator is the following. There are some groups of properties, or genes, that would be advantageous for the offpsring to inherit together from one parent. Such groups of genes, which interact to increase the fitness of the offspring that inherit them, are said to be *coadapted*. For example, if cells A and B are densely connected to each other and parent 1 has the genes (A, x_1, y_1) and (B, x_2, y_2), where (x_1, y_1) and (x_2, y_2) are neighboring locations, it would be advantageous for the offspring to inherit both these genes from one parent so that after crossover cells A and B remain in neighboring locations. If two genes are close to each other in the solution string, they have a lesser probability of being split up when the crossover operator divides the string into two segments. Thus, by shuffling the cells around in the solution string, inversion allows triples of cells that are already well placed relative to each other to be located close to each other in the string. This increases the probability that when the crossover operator splits parent configurations into segments to pass to the offspring, the subplacements consisting of such groups will be passed intact from one parent (or another). This process allows for the formation and survival of highly optimized subplacements long before the optimization of any complete placement is finished. The *inversion rate* is the probability of performing inversion on each individual during each generation. It controls the amount of group formation. Too much inversion will result in the perturbation of the groups already formed.

Selection. After generating offspring, we need a selection procedure in order to choose the next generation from the combined set of parents and offspring. There is a lot of diversity in the selection functions used by various researchers. This section briefly lists some of them. The following sections give the specific functions used in particular algorithms. The three most commonly used selection

methods are competitive, random, and stochastic.

In *competitive selection*, all the parents and offspring compete with each other, and the P fittest individuals are selected, where P is the population size.

In *random selection*, as the name implies, P individuals are selected at random, with uniform probability. Sometimes this is advantageous because that way the population maintains its diversity much longer and the search does not converge to a local optimum. With purely competitive selection, the whole population can quickly converge to individuals that are only slightly different from each other, after which the algorithm will lose its ability to optimize further. (This condition is called premature convergence. Once this occurs, the population will take a very long time to recover its diversity through the slow process of mutation.) A variation of this method is the retention of the best configuration and selection of the rest of the population randomly. This ensures that the fitness will always increase monotonically and we will never lose the best configuration found, simply because it was not selected by the random process.

Stochastic selection is similar to the process described above for the selection of parents for crossover. The probability of selection of each individual is proportional to its fitness. This method includes both competition and randomness.

Comparison with Simulated Annealing. Both simulated annealing and the genetic algorithm are computation intensive. The genetic algorithm, however, has some built-in features, which, if exploited properly, can result in significant savings. One difference is that simulated annealing operates on only one configuration at a time, whereas the genetic algorithm maintains a large population of configurations that are optimized simultaneously. Thus, the genetic algorithm takes advantage of the experience gained in past exploration of the solution space and can direct a more extensive search to areas of lower average cost. Since simulated annealing operates on only one configuration at a time, it has little history to use to learn from past trials.

Both simulated annealing and the genetic algorithm have mechanisms for avoiding entrapment at local optima. In simulated annealing, this is done by occasionally discarding a superior configuration and accepting an inferior one. The genetic algorithm also relies on inferior configurations as a means of avoiding false optima, but since it has a whole population of configurations, the genetic algorithm can keep and process inferior configurations without losing the best ones. Besides, in the genetic algorithm each new configuration is constructed from two previous configurations, which means that in a few iterations, all the configurations in the population have a chance of contributing their good features to form one superconfiguration. In simulated annealing, each new configuration is formed from only one old configuration, which means that the good features of more than one radically different configurations never mix. A configuration is either accepted or thrown away as a whole, depending on its total cost.

On the negative side, the genetic algorithm requires more memory space compared to simulated annealing. For example, a 1000 cell placement problem would require up to 400Kb to store a population of 50 configurations. For moderate sized layout problems, this memory requirement may not pose a significant problem because commercial workstations have 4Mb or more of primary memory. For circuits of the order of 10,000 cells, the genetic algorithm is expected to have a small amount of extra paging overhead compared to simulated annealing, but it is still expected to speed up the optimization due to the efficiency of the search process.

The genetic algorithm is a new and powerful technique. This method depends for its success on the proper choice of the various parameters and functions that control processes like mutation,

selection, and crossover. If the functions are selected properly, a good placement will be obtained. The major problem in devising a genetic algorithm for module placement is choosing the functions most suitable for this problem. A great deal of research is currently being conducted on it. In this section, three algorithms due to Cohoon and Paris [1986], Kling [1987], and Shahookar and Mazumder [1990] are discussed. More work has been done in this field by Chan and Mazumder [1989].

5.1 Genie: Genetic Placement Algorithm

The Genie algorithm was developed by Cohoon and Paris [1986]. The pseudocode is given below:

```
PROCEDURE Genie:
  Initialize;
  N_P ← population size;
  N_O ← N_p P_ψ;
    /* where P_ψ is the desired ratio of the number of offspring to the population size */
  Construct_population(N_P);
  FOR i ← 1 TO N_P score(population[i]);
  ENDFOR;
  FOR i ← 1 TO Number_of_Generations
    FOR j ← 1 TO N_O
      (x, y) ← Choose_Parents;
      offspring[j] ← generate_Offspring(x, y);
      Score(offspring[j]);
    ENDFOR;
    population ← Select(population, offspring, N_P);
    FOR j ← 1 TO N_P
      With probability P_μ Mutate(population[j]);
    ENDFOR;
  ENDFOR;
  Return highest scoring configuration in population;
END.
```

The following is a description of some of the functions used in Cohoon and Paris [1986] and their results.

(1) *Initial population construction.* Cohoon and Paris [1986] proposed two methods for generating the initial population. The first one is to construct the population randomly. The second is to use a greedy clustering technique to place the cells. A net is chosen at random, and the modules connected to it are placed in netlist order. Then, another net connected to the most recently placed module is chosen, and the process is repeated. Experimental observations of Cohoon and Paris show that the initial population constructed by clustering is fitter, but it rapidly converges to a local optimum. Hence, in the final algorithm, they have used a mixed population, a part of which is constructed by each method.

(2) *Scoring function.* The scoring function determines the fitness of a placement. The scoring function σ used in Genie uses the conventional wire length function based on the bounding rectangle. It does not account for cell overlap or row lengths, owing to the gate array layout style. It does, however, account for nonuniform channel usage. This is done as follows: Let

L_i be the perimeter of net i,

r and c be the number of rows and columns, respectively,

h_i be the number of nets intersecting the horizontal channel i,

v_i be the number of nets intersecting the vertical channel i,

s_h be the standard deviation of h_i,

s_v be the standard deviation of v_i,

$\bar{h}$ and $\bar{v}$ be the mean of h_i and v_i respectively,

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

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

Then,

$$\sigma = \frac{1}{2}\sum_{i=1}^{n} L_i + \sum_{i=1}^{r-1} \hat{h}_i^2 + \sum_{i=1}^{c-1} \hat{v}_i^2.$$

This scoring function penalizes all channels that have a wiring density more than one standard deviation above the mean. Thus, it encourages a more uniform distribution of the wiring.

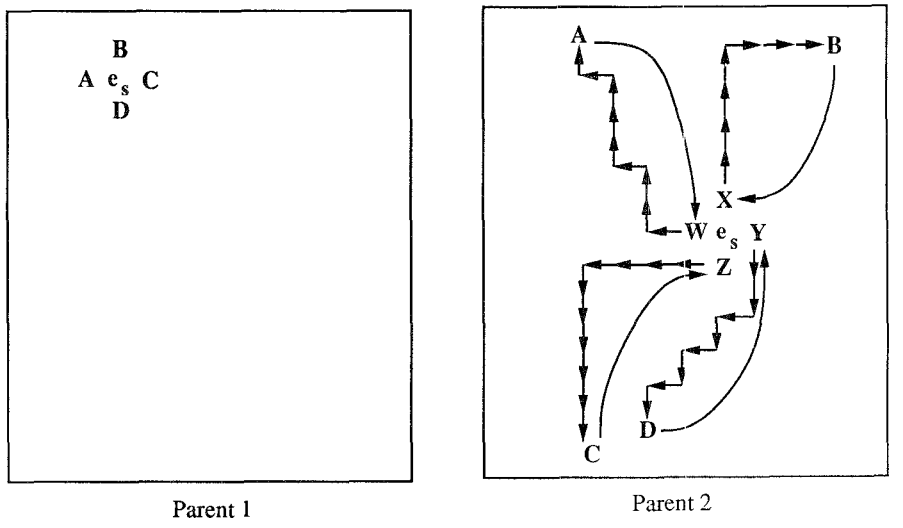
(3) *Parent choice function.* The parent choice function chooses the parent pairs. Four alternatives were considered here. (1) Pair a random string with the fittest string, (2) choose both parents randomly, (3) select parents stochastically, where the probability of each individual being selected as parent is proportional to its fitness, and (4), which is the same as (3) but allows only individuals with above-average fitness to reproduce. The fitness function used here is

$$w(s) = \frac{\sigma_{\max}}{\sigma(s)},$$

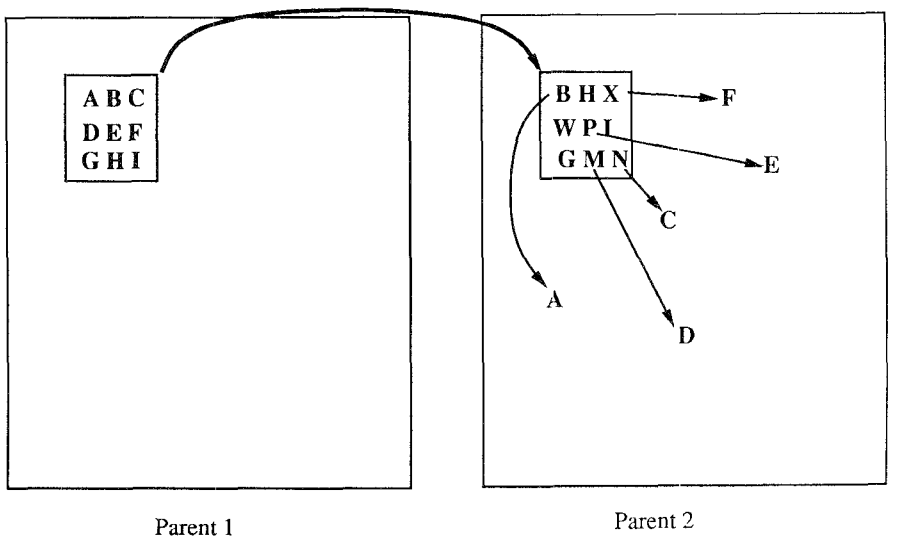
which is equivalent to taking the reciprocal of the cost and normalizing it so that the lowest fitness is 1. If the best configuration is paired with a random one, the population quickly loses diversity and the algorithm converges to a poor local minimum. At the other extreme, if parents are chosen randomly, there is little improvement after several generations and hence no convergence to a good placement. The stochastic functions (3) and (4) produced the best results.

(4) *Crossover operator.* The crossover operator generates offspring from the parents. Two crossover operators have been described. The first selects a random module e_s and brings its four nearest neighbors in parent 1 into neighboring slots in parent 2 while minimally perturbing the other modules. In order to do this, the modules occupying the neighboring locations of e_s in parent 2 are shifted outward by one location at a time in a chain move until a vacant location is found (Figure 31a). The result of this is that a patch consisting of module e_s and its four neighbors is copied from parent 1 into parent 2, and the other modules are shifted by one position in order to make room. The second operator selects a square consisting of $k \times k$ modules from parent 1, where k is a random number with mean 3 and variance 1, and copies it to parent 2. This method tends to duplicate some modules and leave out others. To avoid this conflict, the modules that are in the square patch of parent 2, but not in the patch of parent 1, and that are going to be overwritten are copied to the locations of modules that are in the patch of parent 1 but not in the patch of parent 2, which are thus prevented from being duplicated (Figure 31b). Experiments favor the second operator.

(5) *Selection function.* The selection function determines which configurations will survive into the next generation. The three functions tried are (1) select the best string and $p - 1$ random strings for survival into the next generation, where p is the population size; (2) select p random strings; (3) select strings stochastically, with the probability of selection being proportional to the fitness. The results are similar to those for



(a)



(b)

Figure 31. Crossover in Genie. (a) Crossover operator 1. The modules surrounding ϵ_s in parent 1 are inserted in locations around e_s in parent 2. The displaced modules are shifted one slot at a time so as to cause a minimum disruption in the layout. Thus, parent 2 inherits the $e_s ABCD$ patch from parent 1. (b) Crossover operator 2. Copy the rectangular patch from parent 1 to parent 2. But this would cause the modules *M*, *N*, *P*, *W*, *X*, which are in the segment of parent 2 but not in the segment of parent 1, to be overwritten and lost. Hence, first copy these elements to the locations of *A*, *C*, *D*, *E*, *F*, which are in the segment of parent 1 but not in the segment of parent 2. This would also prevent these modules from being duplicated.

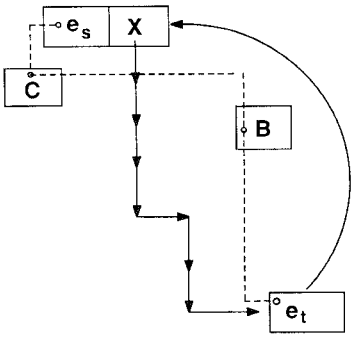


Figure 32. The greedy mutation operator Select the net $e_t BC e_s$ (dotted line) and the target module e_s. Mutate by moving e_t to the location of X, that is, adjacent to e_s, and sliding X one location. Continue sliding the displaced modules until a vacant location is found. This operation reduces the perimeter of the net bounding rectangle.

the parent choice function. If the best configuration and $p - 1$ random ones are chosen, the population quickly loses diversity and converges to a poor local minimum. The function that chooses any p configurations at random or the one that probabilistically favors the choice of the higher scoring configurations perform much better.

(6) *Mutation function.* Two alternatives are (1) perform a series of random interchanges and (2) use a greedy technique to improve the placement cost. The greedy operator chooses a module e_s on a net Z and searches for a module e_t on the same net that is farthest from e_s. e_t is then brought close to e_s, and the displaced modules are shifted one location at a time until a vacant location is found (Figure 32). Thus, the perimeter of the bounding rectangle of net Z is reduced while minimally perturbing the rest of the placement.

Experimental Results. The comparative performance of different variations of the genetic operators has been described above. The algorithm was tried on five circuits with 36–81 cells. The performance was compared against a simulated annealing algorithm also developed by Cohoon and Paris. Genie obtained the same placement quality in two cases and up to 7% worse in the other three cases. The number of configurations examined, however, was only 28% for one circuit, 50% for two circuits, and 75% for two circuits. The actual CPU time was not given.

5.2 ESP: Evolution-Based Placement Algorithm

Kling [1987] and Kling and Bannerjee [1987] developed an evolution-based algorithm that iteratively uses the sequence mutation, evaluation, judgment, and reallocation. The algorithm operates on only one configuration at a time. The modules in the configuration are treated as the population. A mutation is a random change in the placement. Evaluation determines the *goodness* of placement of each module, that is, the individual contribution of the module to the cost. Kling used this measure of goodness instead of the traditional fitness measure in genetic algorithms. The judgment function probabilistically determines whether a module is to be removed and reallocated or not on the basis of its goodness value. In the reallocation phase, all the modules removed during the judgment phase are placed at new locations. The algorithms used for performing these functions are described in detail below.

Mutation. Mutation is done by randomly interchanging a certain number of module pairs without regard to their effect on the placement. The mutation process is controlled by two user-supplied parameters—the probability of occurrence of mutation and the percentage of the total number of modules to be mutated. These two parameters determine the number of mutations performed during each iteration.

Evaluation. Evaluation is a complex process and is the critical step in this algorithm. As mentioned above, it determines the goodness of placement of each module so that the poorly placed modules can be removed for allocation. Kling has

proposed several procedures for evaluating the goodness value.

For gate arrays, the goodness of each module is computed as the ratio of the current value to the precomputed ideal value. The estimate of the current value is based on the product of the connectivity to other modules and the reciprocal of distance from them. An evaluation window consisting of the normalized reciprocal Manhattan distances from the center (called weights) is precomputed as shown in Figure 33a. To evaluate the current value of a module i, the evaluation window is centered over it. For all modules j to which it is connected, the sum

$$r_i = \sum c_{ij} w_j,$$

is calculated, where c_{ij} is the connectivity of the module being evaluated to the jth module and w_j is the weight corresponding to their distance. Figure 33b shows an example of the computation of the current value for a module. The precomputed ideal value is obtained by a similar computation process, but here all the modules connected to the module being evaluated are assumed to be placed in its immediate neighborhood such that the modules with the largest connectivity are placed closest to it (Figure 33c). This is the upper bound on the current value, which would be achieved only by the best-placed modules, which have all connected modules in adjacent positions.

For standard cells, three methods have been proposed. In the first, the concentric circle method (Figure 34), the area of the modules connected to module i is computed. Concentric circles are then defined such that the nth circle covers n times that area. Weights are assigned to the circles from 100% for the innermost circle to 0% for the area outside the outermost circle. The current value of a cell i is determined by the sum

$$r_i = \sum c_{ij} w_j,$$

where c_{ij} is the connectivity with the jth cell and w_j is the weight of the circular region in which the pin of the jth cell is located.

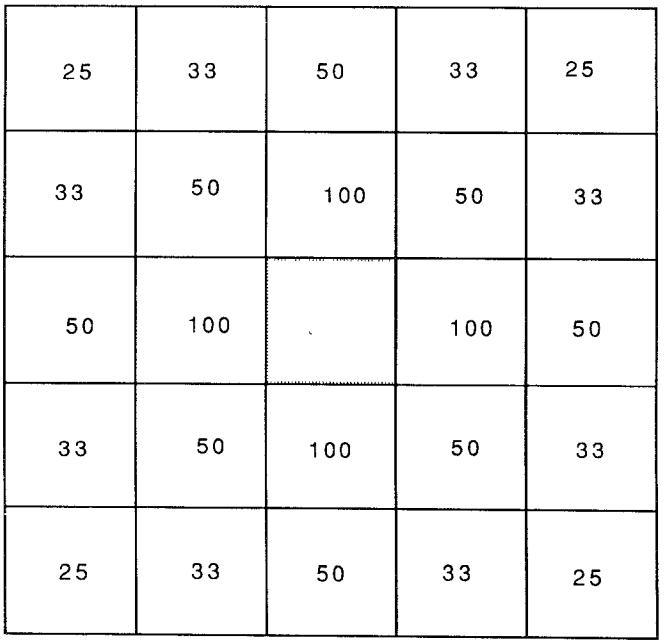
The second evaluation method for standard cells is based on the minimum possible bounding rectangle for a net. The minimum pounding rectangle for each net is computed by placing all modules connected to that net in nearest neighbor locations. To calculate the goodness value of a placed module, its distance from the center of gravity of the net is computed. If it lies within the minimum bounding rectangle, its goodness value is 100%. Otherwise, it is the ratio of the boundary of the net's optimal rectangle to the cell's coordinate closest to the net center.

The third method for standard cells is based on the ratio of the current wire length of the nets connected to a cell to the corresponding optimal wire lengths. The goodness is computed by averaging this ratio for all the nets connected to the cell being evaluated. The result is then normalized in the 0–100% range. This procedure gives the best results for standard cells.

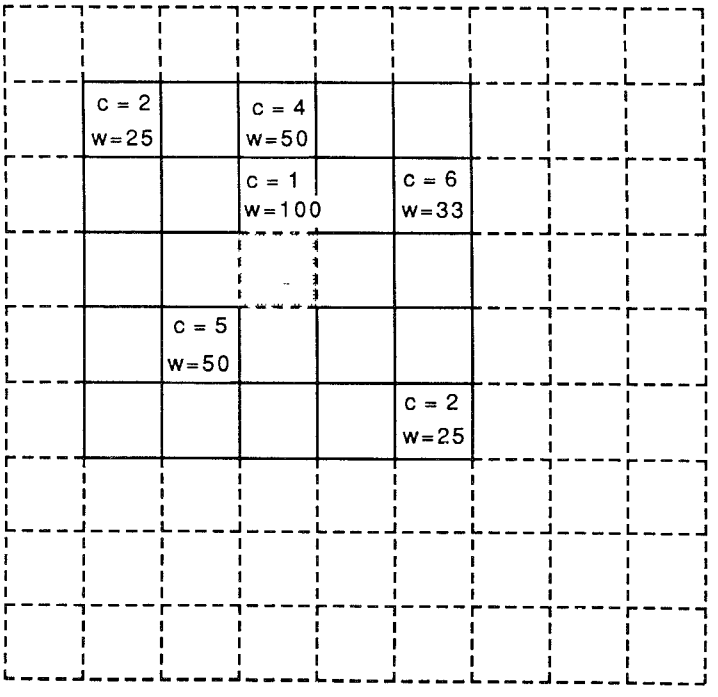
Judgment. In the judgment phase, ill-placed modules are removed for reallocation. The probability of removal of a module increases as its goodness decreases. The goodness of each module is compared to a random number. If the goodness is less than the random number, it is removed.

Reallocation. Reallocation is a critical part of the algorithm. The removed modules should be reallocated at the freed locations so as to improve the placement. Modules to be reallocated are sorted according to their connectivity and placed one at a time. The goodness of each module in all free locations is evaluated. The module is placed at the location giving the best goodness value. Thus, the most densely connected modules get the best choice of location during reallocation.

Preliminary experimental results show this algorithm to be an order of magnitude faster than simulated annealing, with comparable placement quality.

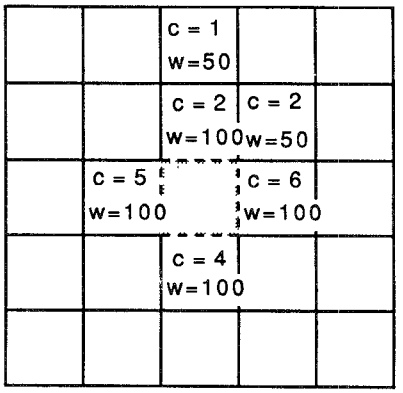


(a)



(b)

Figure 33. Evaluation of goodness value in Kling's algorithm. (a) Evaluation window showing weights of neighboring locations; (b) calculation of current value r; $r = \sum c_i w_i = 2*25 + 4*50 + 1*100 + 6*33 + 5*50 + 2*25 = 848$; (c) calculation of ideal value t. $t = \sum c_i w_i = 1*50 + 2*100 + 2*50 + 5*100 + 6*100 + 4*100 = 1850$. Goodness value: $r/t*100 = 848/1850*100 = 45.83\%$.



(c)

Figure 33. Continued.

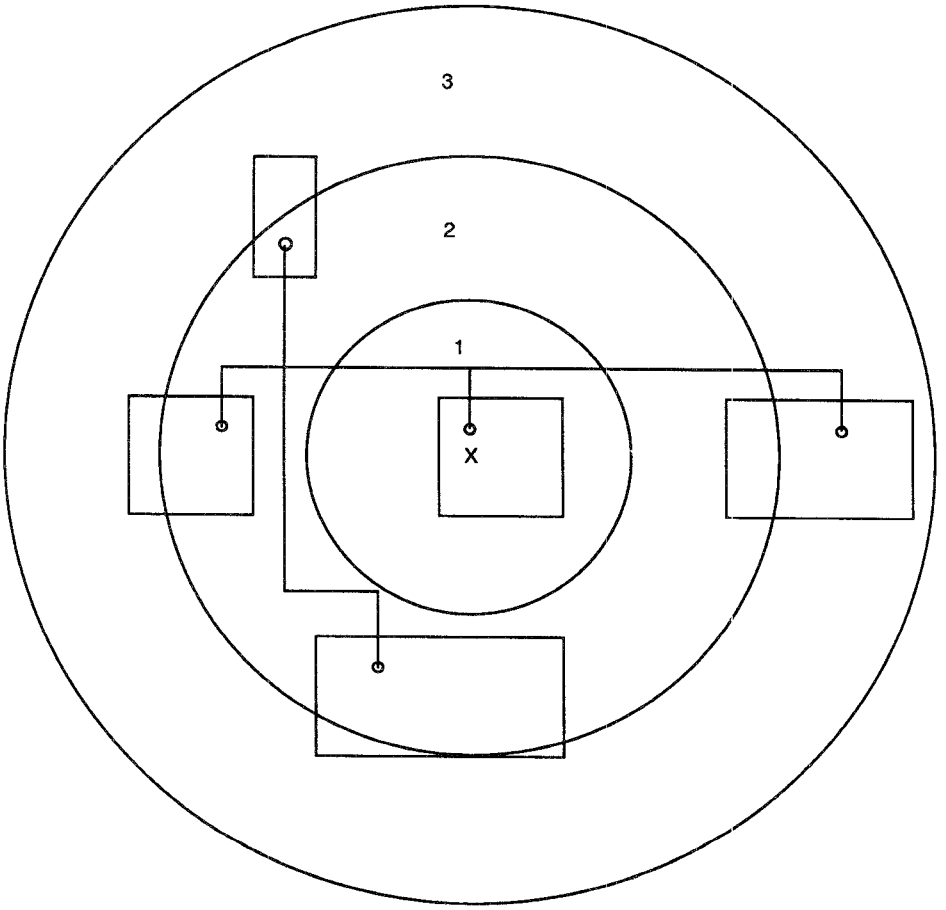


Figure 34. Concentric circle function for the evaluation of goodness of placement of standard cells in Kling's algorithm.

5.3 GASP: Genetic Algorithm for Standard Cell Placement

The authors of this paper have recently implemented a genetic algorithm for cell placement (GASP) [Shahookar and Mazumder 1990] as follows.

5.3.1 Algorithm

The flow chart for GASP is given in Figure 35. First, an initial population of configurations is constructed randomly. Each individual in the population is represented by a set of four integer arrays containing the cell number, the x- and y-coordinates, and a serial number. The coordinates of the cells are determined by placing them end to end in rows. The serial number is used to keep track of the approximate slot in the physical layout area to which each cell is assigned. The population size is provided by the user and determines the tradeoff between processing time and result quality. From experimental observation, it was found that a small population of 24 configurations gave the best performance. Each individual is evaluated to determine its fitness. The fitness is the reciprocal of the wire length. Penalty terms for row length control and cell overlap are not used. Instead, after generating a new configuration, cells are realigned to remove overlap. This is done because removing the overlap takes no more computation time than determining the overlap penalty. Since on average half the cells are moved simultaneously, a majority of the nets are affected. Thus, the wire length has to be computed exhaustively, and no saving is achieved by allowing overlap.

At the beginning of each generation, *inversion* is performed on each individual, with a probability equal to the *inversion rate*. For this purpose, two cut points are determined randomly, and the segment between them in the cell array is flipped, along with the coordinates and the serial numbers (Figure 31). Then crossover takes place. Two individuals are selected from the population at random, with a probability proportional to their fitness. Before crossover, the serial numbers of the second parent are aligned in the same sequence as those of the first parent, so cells in the same array locations correspond to approximately the same locations on the chip. Then segments are exchanged between parents so that for each location on the chip, the child inherits a cell from one parent or another. This process is repeated until the desired number of offspring has been generated. The number of offspring per generation, N_s is determined by the *crossover rate*:

$$N_s = N_p R_c$$

where N_p is the population size and R_c is the crossover rate. Since the number of configurations examined is kept constant, the actual number of generations is increased as the crossover rate is reduced:

$$N_g = \frac{N_{g0} N_p}{N_s},$$

where N_p is the population size and N_{g0} is the number of generations specified by the user.

Each offspring is mutated with a probability equal to the *mutation rate*. Mutation consists of pairwise interchange. Cells in two randomly picked array locations are exchanged, leaving the coordinate arrays unchanged (Figure 30).

After crossover and mutation, the fitness of each offspring is evaluated, and the population for the next generation is selected from the combined set of parents and offspring. Three selection methods have been tried: competitive selection, in which the best of the parents and offspring are selected, random selection, and random selection with the retention of the best individual.

5.3.2 Crossover Operators

Crossover is the primary method of optimization in the genetic algorithm and, in the case of placement, works by combining good subplacements from two different parent placements to generate a new

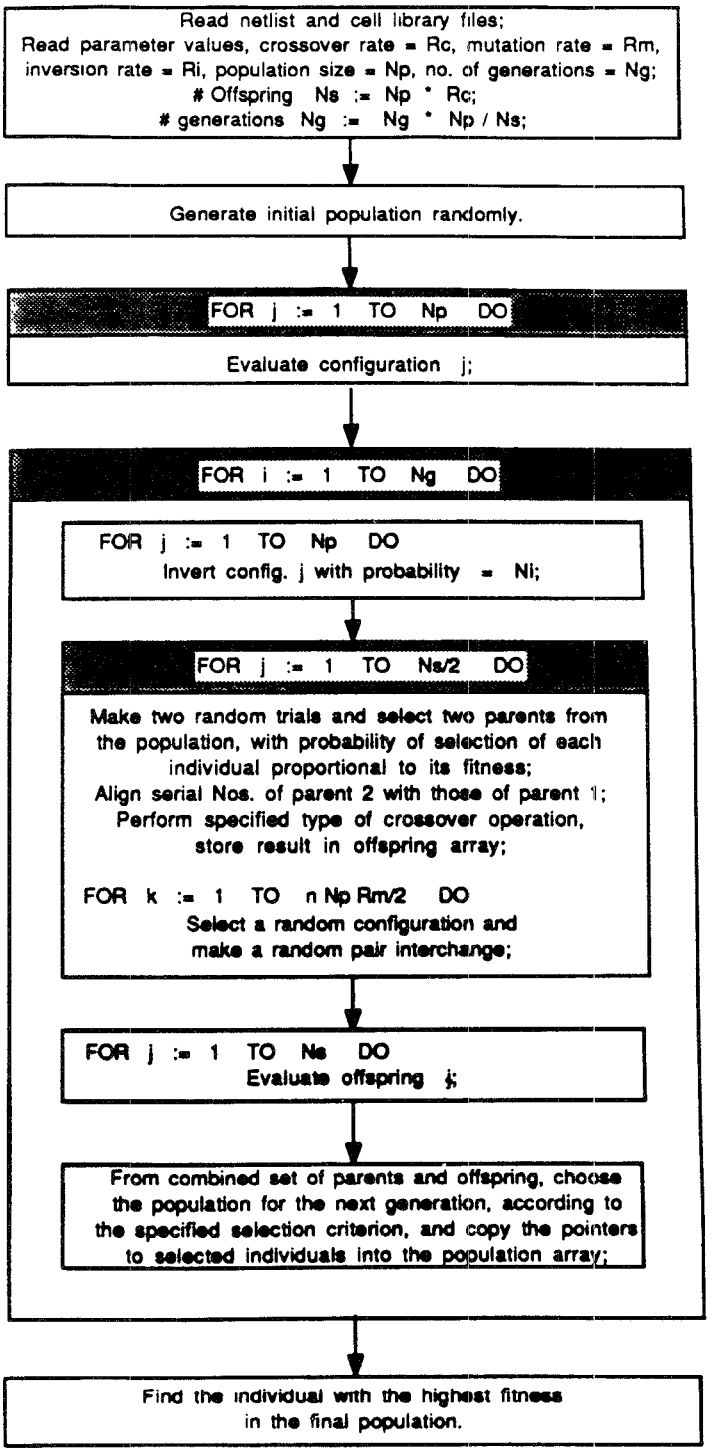


Figure 35. GASP flowchart.

placement. In order to deal with the conflicts that can occur in traditional crossover, one must either find a way to combine two different placements without conflicts or use some method to resolve the conflicts that arise. The performance of three powerful crossover operators have been compared experimentally. Two of them, Order and PMX, differ in their conflict resolution methods, whereas Cycle crossover is a conflictless operator.

Order Crossover. The Order crossover algorithm is as follows. Choose a cut point at random. Copy the array segment to the left of the cut point from one parent to the offspring. Fill the remaining (right) portion of the offspring array by going through the second parent, from the beginning to the end and taking those modules that were left out, in order. An example is shown in Figure 36a. This operator conveys a subplacement from the first parent without any changes, then, to resolve conflicts, compresses the second parent by eliminating the cells conveyed by the first parent and shifting the rest of the cells to the left, without changing their order [Davis 1985]. It then copies this compressed second parent into the remaining part of the offspring array.

PMX. PMX [Goldberg and Lingle 1985] stands for Partially Mapped Crossover. It is implemented as follows. Choose a random cut point and consider the segments following the cut point in both parents as a partial mapping of the cells to be exchanged in the first parent to generate the offspring. Take corresponding cells from the segments of both parents, locate both these cells in the first parent, and exchange them. Repeat this process for all cells in the segment. Thus, a cell in the segment of the first parent and a cell in the same location in the second parent will define which cells in the first parent have to be exchanged to generate the offspring. An example is shown in Figure 36b.

Cycle Crossover. Cycle crossover [Oliver et al. 1987] is an attempt to eliminate the cell conflicts that normally arise in crossover operators. In the offspring generated by cycle crossover, every cell is in the same location as in one parent or the other. For Cycle crossover, we start with the cell in location 1 of parent 1 (or any other reference point) and copy it to location 1 of the offspring. Now consider what will happen to the cell in location 1 of parent 2. The offspring cannot inherit this cell from parent 2, since location 1 in the offspring has been filled. So this cell must be searched in parent 1 and passed on to the offspring from there. Supposing this cell is located in parent 1 at location x. Then it is passed to the offspring at location x. But then the cell at location x in parent 2 cannot be passed to the offspring, so that cell is also passed from parent 1. This process continues until we complete a cycle and reach a cell that has already been passed. Then we choose a cell from parent 2 to pass to the offspring and go through another cycle, passing cells from parent 2 to the offspring. Thus, in alternate cycles, the offspring inherits cells from alternate parents, and the cells are placed in the same locations as they were in the parents from which they were inherited. An example is given in Figure 36c.

5.3.3 Experimental Results

In most cases, either PMX or Cycle crossover performed best, whereas Order crossover performed worst. Cycle crossover was found to be slightly better than PMX. The best compromise of parameters was crossover rate 33%, inversion rate 15%, and mutation rate 0.5%. These values were used in all subsequent experiments.

In all cases, competitive selection of the best of the parents and offspring to be included in the next generation proved to be better than all other strategies. Figures 37a–c show the plots of the lowest and average wiring cost in each generation as the optimization proceeds. The reason for the poor performance of the random selection methods can be clearly seen. Just as it is possible to combine the good features of two parents to form a

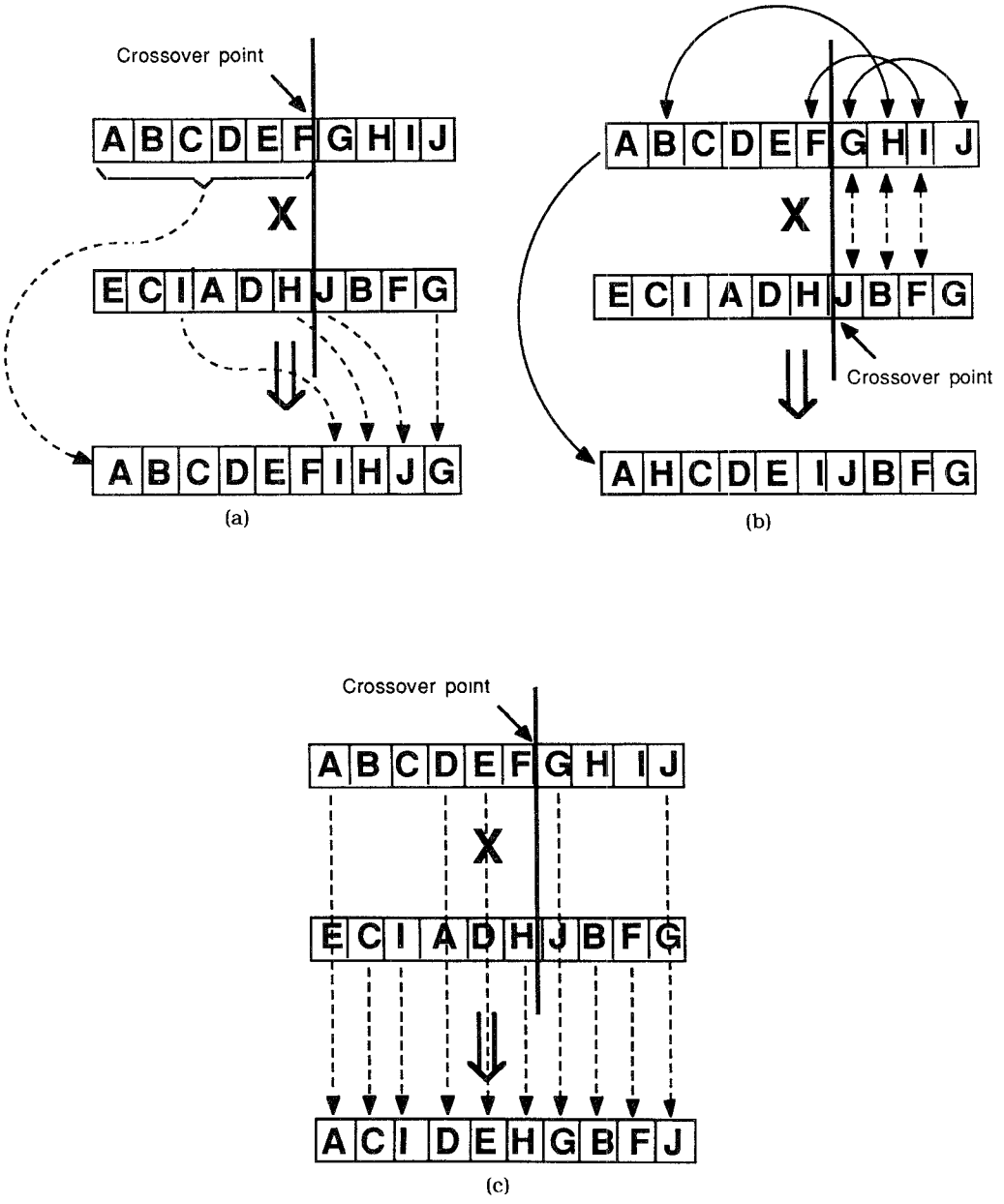


Figure 36. Crossover operators in GASP. (a) Order crossover. Pass the left segment from parent 1. Construct the right segment by taking the remaining cells from parent 2 in the same order. (b) PMX crossover. The right segments of both parents act as a partial mapping of pairwise exchanges to be performed on parent 1. Since the pairs (G, J), (H, B), and (I, F) are situated at the same locations in both parents, exchange these cells in parent 1 to generate the offspring. (c) Cycle crossover. Start by passing A from parent 1 to the offspring. Since E is located at the same position in parent 2, it cannot be passed from there. Therefore, pass E also from parent 1. D is located in the same position in parent 2 as E in parent 1. Therefore, proceed similarly with D. Now A is in the same location, but it has already been processed. This completes a cycle. Start another cycle from parent 2 by passing C to the offspring, and continue by passing B, H, F, and I from parent 2. The third cycle will again be from parent 1 and will pass G and J.

better offspring, it is also possible to combine the bad features to form a far worse offspring. If these offspring are accepted on a random basis, the best and average cost in the population will oscillate, as seen in Figure 37c. The losses involved in the random process far outweigh any advantage gained, and the algorithm takes a much longer time to converge. When we allow for the retention of the best solution along with random selection, the cost of the best solution is seen to decrease monotonically. The average cost of the population still oscillates, however, and the convergence is much slower than for competitive selection.

Comparison with TimberWolf. The performance of the algorithm was compared against TimberWolf 3.3 for five circuits ranging from 100 to 800 cells. It achieved the same quality of placement in about the same amount of CPU time. There are two interesting differences, however.

GASP achieves a very rapid improvement in the beginning, then levels off, as illustrated in Figure 38. On the other hand, for TimberWolf the cost increases for the first few high temperature iterations, and little improvement is achieved during the first half of the run. This means that if a very high-quality placement is not required, GASP will be several times faster.

Another difference is that although the CPU times were comparable, GASP examined 20–50 times fewer configurations for the same quality of placement. This advantage was offset by the excessive evaluation time, which is the bottleneck of the algorithm. In simulated annealing, only two cells are moved at a time, so only a few nets need to be evaluated to determine the change in wire length. In GASP, nearly half the cells are moved simultaneously, and the wire length has to be computed exhaustively. This takes 62–67% of the CPU time, whereas crossover takes only 17% of the time.

6. CONCLUSION

This paper discussed five classes of VLSI module placement algorithms. Simulated annealing is currently the most popular among researchers and is the best algorithm available in terms of the placement quality, but it takes an excessive amount of computation time. It is derived by analogy from the process of annealing, or the attainment of ordered *placement* of atoms in a metal during slow cooling from a high temperature. We discussed the TimberWolf 3.2 algorithm by Sechen, improvements made in TimberWolf 4.2, and other recent developments such as the experiments on the cooling schedule by Nahar et al. [1985] and the Markov chain analysis by Aarts et al.

Min-cut algorithms would rank second in terms of placement quality but would probably be the best in terms of cost/performance ratio, since they are much faster than simulated annealing. These algorithms are based on a simple principle—the groups of cells that are densely connected to each other ought to be placed close together. Thus, by repeated partitioning of the given network to minimize the net cut and each time constraining the subgroups to be placed in different areas in the layout, the wire length is minimized. The algorithms of Breuer have been discussed in this paper, along with improvements such as terminal propagation by Dunlop and Kernighan [1985], and quadrisection by Suaris and Kedem [1987].

Force-directed algorithms operate on the physical analogy of masses connected by springs, where the system would tend to come to rest in the minimum energy state, with minimum tension in the springs, or in terms of the placement problem, the wire length minimized. Force-directed algorithms have been around since the 1960s and were among the first algorithms to be used for placement—mainly printed circuit board placement in those days. A rich variety of implementations have been developed over the years, including constructive (equation solving) methods for determining a minimum-energy configuration from scratch and two types of iterative techniques, one consisting of selecting modules one at a time and determining

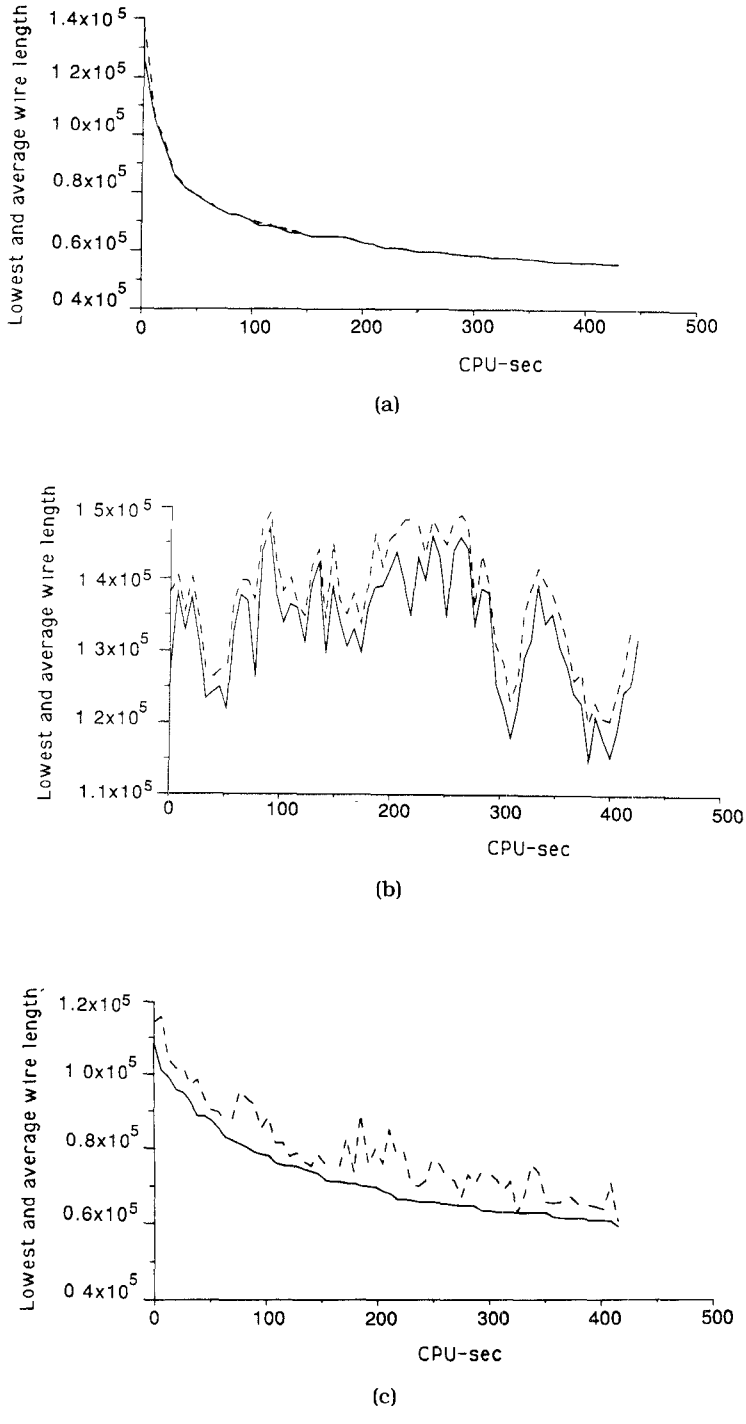


Figure 37. Comparison of selection methods in GASP. (a) Cycle crossover, competitive selection; (b) cycle crossover, random selection; (c) cycle crossover, random + best selection. —, lowest wire length; ---, average wire length.

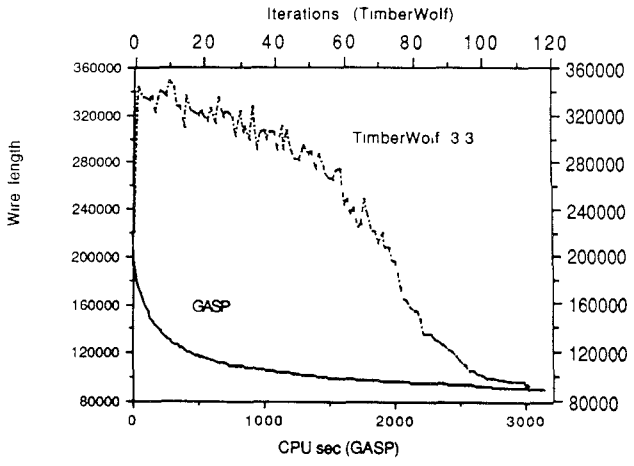


Figure 38. Optimization characteristics of GASP compared to TimberWolf.

an ideal location for them from force considerations and the other consisting of random/exhaustive pairwise interchange, with acceptance of the good moves and rejection of the bad moves, once again on the basis of force considerations. An overview of the various techniques used has been given, along with a sample algorithm and a network example to illustrate the operation of the algorithm. Goto's GFDR algorithm has also been discussed.

Placement is an optimization problem, and methods such as Simplex, Quadratic Programming, and the Penalty Function Method have traditionally been used for various linear and nonlinear optimization problems. Further, the placement problem can also be formulated in terms of the quadratic assignment problem, which can be solved by the eigenvalue method. Accordingly, several papers that use these techniques have been discussed under the category of numerical optimization techniques. The common feature of all these techniques is that they do not constrain the modules to grid points or rows, hence they are more applicable to macroblocks than to standard cells or gate arrays, although the solution generated by numerical techniques can be further processed to map the modules to the nearest grid points.

The final class of algorithms discussed here are genetic algorithms, which, although invented in the 1960s, were not used for placement until 1986. The genetic algorithm is an extremely efficient search and optimization technique for problems with a large and varied search space, as well as problems where more than one physical feature needs to be optimized simultaneously. The genetic algorithm processes a set of alternative placements together and creates a new placement for trial by combining subplacements from two parent placements. This causes the inheritance and accumulation of good subplacements from one generation to the next. It also causes the mixing of the good features of several different placements that are being optimized simultaneously for mutual benefit. Thus, the search through the solution space is inherently parallel. The placement problem is represented in the form of a genetic code, and the genetic operators operate on this code, not directly on the physical layout. This is a major deviation from the conventional placement algorithms that directly apply transformations to the physical layout. This intrinsic parallelism of the genetic algorithm can, however, be a potential problem, and unless a clever representation scheme is devised to represent the

Table 5. Comparison of Placement Algorithms

Algorithm	Result quality	Speed
Simulated annealing	Near optimal	Very slow
Genetic algorithm	Near optimal	Very slow
Force directed	Medium . . . good	Slow . . . medium
Numerical optimization	Medium . . . good	Slow . . . medium
Min-cut	Good	Medium
Clustering and other constructive placement	Poor	Fast

Table 6. Comparison of the Run Times of Placement Algorithms

Implementation	algorithm	No. of cells	CPU hours	Computer hardware	Performance	Reference
Huang et al.	Simulated	469	1.42	VAX 11/780	Wire lengths	[Huang et al. 1986]
TimberWolf 3.2	Annealing	469	3		within ±4%	
Huang		800	10.42			
TimberWolf 3.2		800	10.7			
Dunlop and Kernighan	Min-cut	412	1	VAX 11/780	Comparable to manual layout	[Dunlop and Kernighan 1985]
Quadrisection	Min-cut	173	0.01	VAX 8600	Chip area = 1.11	[Suaris and Kedem 1987]
TimberWolf 3.2		173	0.53		Chip area = 1.0	
Quadrisection		796	0.135		Chip area = 0.91	
TimberWolf 3.2		796	17.8		Chip area = 1.0	
Proud-2	Gauss-	1438	0.014	VAX 8650	Wire length = 0.93	[Tsay et al. 1988]
Proud-4	Seidel	1438	0.027		Wire length = 0.9	
TimberWolf 3.2		1438	2		Wire length = 1.0	
TimberWolf 4.2		1438	0.9		Wire length = 0.84	
Proud-2		3816	0.09		Wire length = 0.90	
Proud-4		3816	0.18		Wire length = 0.91	
TimberWolf 3.2		3816	—		Wire length = 1.0	
TimberWolf 4.2		3816	6.69		Wire length = 0.83	
Proud-2		26277	0.85		Wire length = 1.0	
Proud-4		26277	1.56		Wire length = 0.962	
ESP	Evolution	183	0.43	Sun 3/75	Wire length = 1.0	[Kling 1987]
TimberWolf 3.2		183	2.7		Wire length = 1.0	
GASP	Genetic	469	11.0	Apollo-	Wire length = 1.0	[Shahookar and
TimberWolf 3.2		469	11.3	DN4000	Wire length = 1.02	Mazumder 1990]
GASP		800	12.5		Wire length = 1.0	
TimberWolf 3.2		800	13.7		Wire length = 0.87	

physical placement as a genetic code, the algorithm may prove ineffective. In this paper, three implementations of the genetic algorithm that overcome these problems in different ways were described.

Table 5 is an approximate comparison of the performance of the algorithms discussed here. Table 6 gives the run time and performance of some of the algorithms. The wire length or chip area in the performance column has been normalized. This data can only give partial comparisons, since different papers have reported results on different circuits and have used different computer hardware. An attempt has been made to group the data according to the computer hardware used.

Despite the bewildering variety of algorithms available, efficient module placement has so far remained an elusive goal. Most of the heuristics that have been tried take excessive amounts of CPU

time and produce suboptimal results. Until recently excessive computation times had prohibited the processing of circuits with more than a few thousand modules. As fast simulated annealing and min-cut algorithms discussed above are cast into fully developed place and route packages, however, this situation is expected to change. Preliminary results show that these algorithms have the capability to produce near-optimal placements in reasonable computation time.

The following is a list of other surveys and tutorials on cell placement in chronological order: Hanan and Kurtzberg [1972a], Preas [1979], Soukup [1981], Cheng [1984], Hu and Kuh [1985], Hildebrandt [1985], Goto and Matsuda [1986], Preas and Karger [1986], Sangiovanni-Vincentelli [1987], Wong et al. [1988], and Preas and Lorenzetti [1988].

Robson [1984] and VLSI [1987, 1988] list exhaustive surveys on commercially available automatic layout software. These surveys indicate that force-directed placement was the algorithm of choice in systems available in 1984 [Robson 1984]. In 1987 and 1988, we see an even mix of force-directed algorithms, min-cut, and simulated annealing [VLSI 1987, 1988]. According to the 1988 survey, a few of these systems can be used to place and route sea-of-gates arrays with more than 100,000 gates, in triple metal, using up to 80% of the available gates [VLSI 1988]. Another trend immediately obvious from these surveys is that almost all the systems can be run on desktop workstations—Sun, Apollo, or MicroVAX. Thus automated layout systems are very widely available. They have made it possible to transfer the task of designing and laying out custom ICs from the IC manufacturer to the client.

ACKNOWLEDGMENTS

This research was partially supported by the NSF Research Initiation Awards under the grant number MIP-8808978, the University Research Initiative program of the U.S. Army under the grant number DAAL 03-87-K-0007, and the Digital Equipment Corporation Faculty Development Award. K. Shahookar is supported by the Science and Technology Scholarship Program of the Government of Pakistan.

REFERENCES

AARTS, E. H. L , DEBONT, F. M. J., AND HABERS, E. H. A. 1985. Statistical cooling: A general approach to combinatorial optimization problems. *Philips J. Res. 40*, 4, 193-226.

AARTS, E. H. L., DEBONT, F. M. J., AND HABERS, E. H. A. 1986. Parallel implementations of the statistical cooling algorithm. *Integration, VLSI J. 4*, 3 (Sept.) 209-238.

AKERS, S. B. 1981. On the use of the linear assignment algorithm in module placement. In *Proceedings of the 18th Design Automation Conference*. pp. 137-144.

ANTREICH, K. J., JOHANNES, F. M., AND KIRSCH, F H. 1982. A new approach for solving the placement problem using force models. In *Proceedings of the IEEE International Symposium on Circuits and Systems*. pp. 481-486.

BANNERJEE, P., AND JONES, M. 1986. A parallel simulated annealing algorithm for standard cell placement on a hypercube computer. In *Proceedings of the IEEE International Conference on Computer Design*. p. 34.

BENDERS, J. F. 1962. Partitioning procedures for solving mixed variable problems. *Numer. Math. 4*, 238-252.

BLANKS, J. P. 1984. Initial placement of gate arrays using least squares methods. In *Proceedings of the 21st Design Automation Conference*. pp. 670-671.

BLANKS, J. P. 1985a. Near-optimal placement using a quadratic objective function. In *Proceedings of the 22nd Design Automation Conference*. pp. 609-615.

BLANKS, J. P. 1985b. Use of a quadratic objective function for the placement problem in VLSI design. Ph.D. dissertation, Univ. of Texas at Austin.

BREUER, M. A. 1977a. Min-cut placement. *J. Design Automation and Fault-Tolerant Computing 1*, 4 (Oct.) 343-382.

BREUER, M. A. 1977b. A class of min-cut placement algorithms. In *Proceedings of the 14th Design Automation Conference*. pp. 284-290

CASSOTO, A., ROMEO, F., AND SANGIOVANNI-VINCENTELLI, A. 1987. A parallel simulated annealing algorithm for the placement of standard cells. *IEEE Trans. Comput.-Aided Design CAD-6*, 5 (May), 838.

CHAN, H. M., AND MAZUMDER, P. 1989. A genetic algorithm for macro cell placement. Tech. Rep. Computing Research Laboratory, Dept. of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, Mich.

CHANG, S. 1972. The generation of minimal trees with a steiner topology. *J. ACM 19*, 4 (Oct.), 699–711.

CHEN, N. P. 1983. New algorithms for steiner tree on graphs. In *Proceedings of the International Symposium on Circuits and Systems*. pp. 1217–1219.

CHENG, C. 1984. Placement algorithms and applications to VLSI design. Ph.D. dissertation Dept. of Electrical Engineering, Univ. of California, Berkeley.

CHENG, C., AND KUH, E. 1984. Module placement based on resistive network optimization. *IEEE Trans. Comput.-Aided Design CAD-3*, 7 (July), 218–225.

CHUNG, M. J., AND RAO, K. K. 1986. Parallel simulated annealing for partitioning and routing. In *Proceedings of the IEEE International Conference on Computer Design*. pp. 238–242.

CHYAN, D., AND BREUER, M. A. 1983. A placement algorithm for array processors. In *Proceedings of the 20th Design Automation Conference*. pp. 182–188.

COHOON, J. P., AND SAHNI, S. 1983. Heuristics for the board permutation problem. In *Proceedings of the 20th Design Automation Conference*.

COHOON, J. P., AND PARIS, W. D. 1986. Genetic placement. In *Proceedings of the IEEE International Conference on Computer-Aided Design*. pp. 422–425.

CORRIGAN, L. I. 1979. A placement capability based on partitioning. In *Proceedings of the 16th Design Automation Conference*. pp. 406–413.

DAVIS, L. 1985. Applying adaptive algorithms to epistatic domains. In *Proceedings of the International Joint Conference on Artificial Intelligence*.

DONATH, W. E. 1980. Complexity theory and design automation. In *Proceedings of the 17th Design Automation Conference*. pp. 412–419.

DUNLOP, A. E., AND KERNIGHAN, B. W. 1985. A procedure for placement of standard cell VLSI circuits. *IEEE Trans. Comput.-Aided Design CAD-4*, 1 (Jan.), 92–98.

FIDUCCIA, C. M., AND MATTHEYSES, R. M. 1982. A linear-time heuristic for improving network partitions. In *Proceedings of the 19th Design Automation Conference*. pp. 175–181.

FISK, C. J., CASKEY, D. L., AND WEST, L. E. 1967. Accel: Automated circuit card etching layout. *Proc. IEEE 55*, 11 (Nov.) 1971–1982.

FUKUNAGA, K., YAMADA, S., STONE, H., AND KASAI, T. 1983. Placement of circuit modules using a graph space approach. In *Proceedings of the 20th Design Automation Conference*. 465–473.

GIDAS, B. 1985. Non-stationary Markov chains and convergence of the annealing algorithm. *J. Stat. Phys. 39*, 73–131.

GILMORE, P. C. 1962. Optimum and suboptimum algorithms for the quadratic assignment problem. *J. SIAM 10*, 2 (June), 305–313.

GOLDBERG, D. E., AND LINGLE, R. 1985. Alleles, loci and the traveling salesman problem. In *Proceedings of the International Conference on Genetic Algorithms and their Applications*.

GOTO, S. 1981. An efficient algorithm for the two-dimensional placement problem in electrical circuit layout. *IEEE Trans. Circuits Syst., CAS-28* (Jan.), 12–18.

GOTO, S., AND KUH, E. S. 1976. An approach to the two-dimensional placement problem in circuit layout. *IEEE Trans. Circuits Syst. CAS-25*, 4, 208–214.

GOTO, S., CEDERBAUM, I., AND TING, B.S. 1977. Suboptimal solution of the backboard ordering with channel capacity constraint. *IEEE Trans. Circuits Syst.* (Nov. 1977), 645–652.

GOTO, S., AND MATSUDA, T. 1986. Partitioning, assignment and placement. In *Layout Design And Verification*, T. Ohtsuki, Ed. Elsevier North-Holland, New York, Chap. 2, pp. 55–97.

GREENE, J. W., AND SUPOWIT, K. J. 1984. Simulated annealing without rejected moves. In *Proceedings of the IEEE International Conference on Computer Design*. pp. 658–663.

GREFENSTETTE, J. J., Ed. 1985. In *Proceedings of an International Conference on Genetic Algorithms and their Applications*. Pittsburgh, Penn.

GREFENSTETTE, J. J., Ed. 1987. In *Proceedings of the 2nd International Conference on Genetic Algorithms and their Applications*. Cambridge, Mass.

GROVER, L. K. 1987. Standard cell placement using simulated sintering. In *Proceedings of the 24th Design Automation Conference*. pp. 56–59.

HAJEK, B. 1988. Cooling schedules for optimal annealing. *Oper. Res. 13*, 2 (May), 311–329.

HALL, K. M. 1970. An *r*-dimensional quadratic placement algorithm. *Manage. Sci. 17*, 3 (Nov.), 219–229.

HANAN, M., AND KURTZBERG, J. M. 1972a. Placement techniques. In *Design Automation of Digital Systems, 1*, M. A. Breuer, Ed. Prentice Hall, Englewood Cliffs, N.J., Chap. 5, pp. 213–282.

HANAN, M., AND KURTZBERG, J. M. 1972b. A review of placement and quadratic assignment problems. *SIAM Rev. 14*, 2 (Apr.), 324–342.

HANAN, M., AND WOLFF, P. K., AND AGULE, B. J. 1976a. Some experimental results on placement techniques. In *Proceedings of the 13th Design Automation Conference*. pp. 214–224.

HANAN, M., AND WOLFF, P. K., AND AGULE, B. J. 1976b. A study of placement techniques. *J. Design Automation and Fault-Tolerant Computing 1*, 1 (Oct.), 28–61.

HANAN, M., WOLFF, P. K., AND AGULE, B. J. 1978. Some experimental results on placement

techniques. *J. Design Automation and Fault-Tolerant Computing 2* (May), 145-168.

HERRIGEL, A., AND FICHTNER, W. 1989. An analytic optimization technique for placement of macrocells. In *Proceedings of the 26th Design Automation Conference*. pp. 376-381.

HILDEBRANDT, T. 1985. An annotated placement bibliography. *ACM SIGDA Newsletter 15*, 4 (Dec.), 12-21.

HILLNER, H., WEIS, B. X., AND MLYNSKI, D. A. 1986. The discrete placement problem: A dynamic programming approach. In *Proceedings of the International Symposium on Circuits and Systems*. pp. 315-318.

HOLLAND, J. H. 1975. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, Mich.

HU, T. C., AND KUH, E. S. 1985. *VLSI Circuit Layout*. IEEE Press, New York.

HUANG, M. D., ROMEO, F., AND SANGIOVANNI-VINCENTELLI, A. 1986. An efficient general cooling schedule for simulated annealing. In *Proceedings of the IEEE International Conference on Computer-Aided Design*. pp. 381-384.

HWANG, F. K. 1976. "On Steiner Minimal Trees with Rectilinear Distance," *SIAM J. Appl. Math*. Vol. 30, pp.104-114, 1976.

HWANG, F. K. 1979. An $O(n \log n)$ algorithm for suboptimal rectilinear steiner trees. *IEEE Trans. Circuits Syst. CAS-26*, 1, 75-77.

JARMON, D. 1987. An automatic placer for arbitrary sized rectangular blocks based on a cellular model. In *Proceedings of the IEEE International Conference on Computers and Applications*. pp. 842-845.

JOHANNES, F. M., JUST, K. M., AND ANTREICH, K. J. 1983. On the force placement of logic arrays. In *Proceedings of the 6th European Conference on Circuit Theory and Design*. pp. 203-206.

JOHNSON, D. B., AND MIZOGUCHI, T. 1978. Selecting the kth element in $X + Y$ and $X_1 + X_2 + \cdots + X_m$. *SIAM J. Comput. 7*, 2 (May), 141-143.

KAMBE, T., CHIBA, T., KIMURA, S., INUFUSHI, T., OKUDA, N., AND NISHIOKA, I. 1982. A placement algorithm for polycell LSI and its evaluation. In *Proceedings of the 19th Design Automation Conference*. pp. 655-662.

KANG, S. 1983. Linear ordering and application to placement. In *Proceedings of the 20th Design Automation Conference*. pp. 457-464.

KAPPEN, H. J., AND DE BONT, F. M. J. 1990. An efficient placement method for large standard-cell and sea-of-gates designs. In *Proceedings of the IEEE European Design Automation Conference*. pp. 312-316.

KARGER, P. G., AND MALEK, M. 1984. Formulation of component placement as a constrained optimization problem. In *Proceedings of the International Conference on Computer Design*. pp. 814-819.

KERNIGHAN, B. W., AND LIN, S. 1970. An efficient heuristic procedure for partitioning graphs. *Bell Syst. Tech. J. 49*, 2, 291-308.

KIRKPATRICK, S., GELATT, C. D., AND VECCHI, M. P. 1983. Optimization by simulated annealing. *Science 220*, 4598 (May), 671-680.

KLING, R. M. 1987. Placement by simulated evolution. Master's thesis, Coordinated Science Lab, College of Engr., Univ. of Illinois at Urbana-Champaign.

KLING, R., AND BANNERJEE, P. 1987. ESP: A new standard cell placement package using simulated evolution. In *Proceedings of the 24th Design Automation Conference*. pp. 60-66.

KOZAWA, T., MIURA, T., AND TERAI, H. 1984. Combine and top down block placement algorithm for hierarchical logic VLSI layout. In *Proceedings of the 21st Design Automation Conference*. pp. 535-542.

KOZAWA, T , TERAI, H., ISHII, T., HAYASE, M., MIURA, C., OGAWA, Y., KISHIDA, K., YAMADA, N., AND OHNO, Y. 1983. Automatic placement algorithms for high packing density VLSI. In *Proceedings of the 20th Design Automation Conference*. pp. 175-181.

KRUSKAL, J. 1956. On the shortest spanning subtree of a graph and the traveling salesman problem. In *Proceedings of the American Mathematical Society*, Vol. 7, No. 1, pp. 48-50.

VAN LAARHOVEN, P. J. M., AND AARTS, E. H. L. 1987. *Simulated Annealing: Theory and Applications*. D. Riedel, Dordrecht-Holland.

LAM, J., AND DELOSME, J. 1986. Logic minimization using simulated annealing. In *Proceedings of the IEEE International Conference on Computer-Aided Design*. p. 378.

LAM, J., AND DELOSME, J. 1988. Performance of a new annealing schedule. In *Proceedings of the 25th Design Automation Conference*. pp. 306-311.

LAUTHER, U. 1979. A min-cut placement algorithm for general cell assemblies based on a graph representation. In *Proceedings of the 16th Design Automation Conference*. pp. 1-10.

LEIGHTON, F. T. 1983. *Complexity Issues in VLSI*. MIT Press, Cambridge, Mass.

LUNDY, M., AND MEES, A. 1984. Convergence of the annealing algorithm. In *Proceedings of the Simulated Annealing Workshop*.

MAGNUSON, W. G. 1977. A comparison of constructive placement algorithms. *IEEE Region 6 Conf. Rec.* 28-32.

MALLELA, S., AND GROVER, L. K. 1988. Clustering based simulated annealing for standard cell placement. In *Proceedings of the 25th Design Automation Conference*. pp. 312-317.

MARKOV, L. A., FOX, J. R., AND BLANK, J. H. 1984. Optimization technique for two-dimensional placement. In *Proceedings of the 21st Design Automation Conference*. pp. 652-654.

MITRA, D., ROMEO, F., AND SANGIOVANNI-VINCENTELLI, A. 1985. Convergence and finite-time behavior of simulated annealing. In *Proceedings of the 24th Conference on Decision and Control*. pp. 761–767.

MOGAKI, M., MIURA, C., AND TERAI, H. 1987. Algorithm for block placement with size optimization technique by the linear programming approach. In *Proceedings IEEE International Conference on Computer-Aided Design*. pp. 80–83.

MUROGA, S. 1982. *VLSI System Design*. John Wiley, New York, Chap. 9, pp. 365–395.

NAHAR, S., SAHNI, S., AND SHRAGOWITZ, E. 1985. Experiments with simulated annealing. In *Proceedings of the 22th Design Automation Conference*. pp. 748–752.

OLIVER, I. M., SMITH, D. J., AND HOLLAND, J. R. C. 1985. A study of permutation crossover operators on the traveling salesman problem. In *Proceedings of the International Conference on Genetic Algorithms and their Applications*. pp. 224–230.

OTTEN, R., AND VAN GINNEKIN, L. 1984. Floorplan design using simulated annealing. In *Proceedings of the IEEE International Conference on Computer-Aided Design*. pp. 96–98.

PALCZEWSKI, 1984. Performance of algorithms for initial placement. In *Proceedings of the 21st Design Automation Conference*. pp. 399–404.

PERSKY, G., DEUTSCH, D. N., AND SCHWEIKERT, D. J., 1976. LTX: A system for the directed automatic design of LSI circuits. In *Proceedings of the 13th Design Automation Conference*. pp. 399–407.

PREAS, B. T. 1979. Placement and routing algorithms for hierarchical integrated circuit layout. Ph.D. dissertation, Dept. of Electrical Engr., Stanford Univ. Also Tech. Rep. 180, Computer Systems Lab, Stanford Univ.

PREAS, B. T., AND KARGER, P. G. 1986. Automatic placement: A review of current techniques. In *Proceedings of the 23rd Design Automation Conference*. pp. 622–629.

PREAS, B., AND LORENZETTI, M. 1988. Placement, assignment and floorplanning. In *20Physical Design Automation of VLSI Systems*. The Benjamin Cummings Publishing Co., Menlo Park, Calif., Chap. 4, pp. 87–156.

QUINN, N. R. 1975. The placement problem as viewed from the physics of classical mechanics. In *Proceedings of the 12th Design Automation Conference*. pp. 173–178.

QUINN, N. R., AND BREUER, M. A. 1979. A force directed component placement procedure for printed circuit boards. *IEEE Trans. Circuits Syst. CAS-26* (June), 377–388.

RANDELMAN, R. E., AND GREST, G. S. 1986. N-city traveling salesman problem: Optimization by simulated annealings. *J. Stat. Phys. 45*, 885–890.

ROBSON, G. 1984. Automatic placement and routing of gate arrays. *VLSI Design 5*, 4, 35–43.

ROMEO, F., AND SANGIOVANNI-VINCENTELLI, A. 1985. Convergence and finite time behavior of simulated annealing. In *Proceedings of the 24th Conference on Decision and Control*. pp. 761–767.

ROMEO, F., SANGIOVANNI-VINCENTELLI, A., AND SECHEN, C. 1984. Research on simulated annealing at Berkeley. In *Proceedings of the IEEE International Conference on Computer Design*. pp. 652–657.

SAHNI, S., AND BHATT, A. 1980. The complexity of design automation problems. In *Proceedings of the 17th Design Automation Conference*. pp. 402–411.

SANGIOVANNI-VINCENTELLI, A. 1987. Automatic layout of integrated circuits. In *Design Systems for VLSI Circuits*, G. De Micheli, A. Sangiovanni-Vincentelli, and P. Antognetti, Eds. Kluwer Academic Publishers, Hingham, Mass., pp. 113–195.

SCHWEIKERT, D. G. 1976 "A 2-dimensional placement algorithm for the layout of electrical circuits. In *Proceedings of the Design Automation Conference*. pp. 408–416.

SCHWEIKERT, D. G., AND KERNIGHAN, B. W. 1972. A proper model for the partitioning of electrical circuits. In *Proceedings of the 9th Design Automation Workshop*. pp. 57–62.

SECHEN, C. 1986. *The TimberWolf3.2 Standard Cell Placement and Global Routing Program*. User's Guide for Version 3.2, Release 2.

SECHEN, C. 1988a. Chip-planning, placement, and global routing of macro/custom cell integrated circuits using simulated annealing. In *Proceedings of the Design Automation Conference*. pp. 73–80.

SECHEN, C. 1988b. *VLSI Placement and Global Routing Using Simulated Annealing*. Kluwer, B. V., Deventer, The Netherlands.

SECHEN, C. AND LEE, K.-W. 1987. An improved simulated annealing algorithm for row-based placement. In *Proceedings of the IEEE International Conference on Computer-Aided Design*. pp. 478–481.

SECHEN, C., AND SANGIOVANNI-VINCENTELLI, A. 1986. TimberWolf3.2: A new standard cell placement and global routing package. In *Proceedings of the 23rd Design Automation Conference*. pp. 432–439.

SHA, L. AND BLANK, T. 1987. ATLAS: A technique for layout using analytic shapes. In *Proceedings of the IEEE International Conference on Computer-Aided Design*. pp. 84–87.

SHA, L., AND DUTTON, R. 1985. An analytical algorithm for placement of arbitrarily sized rectangular blocks. In *Proceedings of the 22nd Design Automation Conference*. pp. 602–607.

SHAHOOKAR, K., AND MAZUMDER, P. 1990. A genetic approach to standard cell placement

using meta-genetic parameter optimization. *IEEE Trans. Comput.-Aided Design 9*, 5 (May), 500-511.

SHIRAISHI, H., AND HIROSE, F. 1980. Efficient placement and routing techniques for master-slice LSI In *Proceedings of the 17th Design Automation Conference*. pp. 458-464.

SOUKUP, J. 1981. Circuit layout. *Proc. IEEE 69*, 10 (Oct.), 1281-1304.

STEINBERG, L. 1961. The backboard wiring problem: A placement algorithm. *SIAM Rev. 3*, 1 (Jan.), 37-50.

STEVENS, J. E. 1972. Fast heuristic techniques for placing and wiring printed circuit boards. Ph.D. dissertation, Comp. Sci. Dept., Univ. of Illinois.

SUARIS, P., AND KEDEM, G. 1987. Quadrisection: A new approach to standard cell layout. In *Proceedings of the IEEE International Conference on Computer-Aided Design*. pp. 474-477

SZU, H. 1986. Fast simulated annealing. In *Proceedings of the AIT Conference. Neural Networks for Computing*. pp. 420-425.

TSAY, R., KUH, E. AND HSU, C. 1988. Module placement for large chips based on sparse linear equations. *Int. J. Circuit Theory Appl.* 16, 411-423.

UEDA, K., KASAI, R., AND SUDO, T. 1986. Layout strategy, standardization, and CAD tools. In *Layout Design And Verification*, T. Ohtsuki, Ed. Elsevier Science Pub. Co., New York, Chap. 1.

VECCHI, M. P., AND KIRKPATRICK, S. 1983. Global wiring by simulated annealing. *IEEE Trans. Comput.-Aided Design CAD-2*, 215-222.

VLSI SYSTEMS DESIGN STAFF. 1987. Survey of automatic layout software. *VLSI Syst. Design 8*, 4, 78-89.

VLSI SYSTEMS DESIGN STAFF. 1988. Survey of automatic IC layout software. *VLSI Syst. Design 9*, 4, 40-49

WALSH, G. R. 1975. *Methods of Optimization*. John Wiley and Sons, New York.

WHITE, S. R. 1984. Concepts of scale in simulated annealing. In *Proceedings of the IEEE International Conference on Computer Design*. pp. 646-651

WIPFLER, G. J., WIESEL, M., AND MLYNSKI, D. A. 1982. A combined force and cut algorithm for hierarchical VLSI layout. In *Proceedings of the 19th Design Automation Conference*. pp. 671-677.

WONG, D. F., LEONG, H. W., AND LIU, C. L. 1986. Multiple PLA folding by the method of simulated annealing In *Proceedings of the Custom IC Conference*. pp. 351-355.

WONG, D. F., LEONG, H. W., AND LIU, C. L. 1988. Placement. In *Simulated Annealing for VLSI Design*, Kluwer B.V., Deventer, The Netherlands, Chap. 2.

Received July 1988, final revision accepted April 1990