Greedy Algorithms for VLSI Placement and Routing

by

Mosur Kumaraswamy Mohan

Computer Sciences Technical Report #716
September 1987

GREEDY ALGORITHMS FOR VLSI MODULE PLACEMENT AND ROUTING

by

MOSUR KUMARASWAMY MOHAN

A thesis submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN-MADISON

1987

ABSTRACT

GREEDY ALGORITHMS FOR VLSI MODULE PLACEMENT AND ROUTING

This thesis addresses the problem of placing a set of rectangular *modules* on a chip, and routing wires between numbered *terminals* on the module-boundaries. The aim is to minimize the total area and wire-length of the final layout. This problem belongs to the class of NP-hard problems, for which no optimum solutions exist. Approximate solutions by linear programming approaches are useful in restricted problems, where all modules have a uniform size and aspect ratio, but fail in the general case. Other approaches such as bipartitioning or clustering are either computationally expensive, or produce unsatisfactory layouts. We concentrate our work on the development of the *greedy approach* to the placement and routing problems, its synthesis with a clustering technique, and on its combination with a novel *incremental global router*.

The greedy approach attempts to build a layout by making placement and orientation decisions with limited information on hand. The conventional scheme of a separate placement and wire-routing phase is replaced by an incremental process. Modules are selected in a sequence dictated by a clustering scheme, and are placed by a recursive module-building algorithm. Global routing is carried out incrementally as a sub-process within the module-builder. Every partial layout can thus be treated as a module, with its internal wiring completed to allow only external nets to appear on its rectangular boundary. This also guarantees completion of routing, unlike conventional schemes. The absence of backtracking makes this a computationally fast approach.

The greedy placer and incremental global router was implemented and tested on sample circuits published in the literature. The resulting layouts had areas and wirelengths that were either close to those of the original examples, or, in many cases, better than the originals. The tests also studied the effects of varying the sequence of module selection for placement, the size of the clusters, etc. Guidelines were established for determining the factors that influence such variations in the use of the Placer. The greedy approach, therefore, was successful in performing as well as or better than conventional approaches, at much lower computational as well as development costs.

TABLE OF CONTENTS

LIST OF FIGURES AND TABLES

Chapter 1

INTRODUCTION

With the widespread use of very large-scale integrated circuits (or *VLSI* circuits) in almost every piece of computing machinery, the design and implementation of circuits has become a complex and demanding task. This area of modern technology has seen dramatic progress in the last decade. Practically every aspect of circuit design has experienced an explosive growth in the size and complexity of the problems that need to be solved [Mead80a]. One of the primary goals of research and development in circuit design and implementation has been to devise algorithms that make it possible to manufacture cheaper and faster chips.

The advantages of mass production and economies of scale are especially evident in the field of VLSI chip manufacture. The exact technology of manufacture of VLSI circuits is irrelevant to our discussion; suffice it to know that a chip is manufactured by exposing certain areas of silicon to certain chemicals, producing active electrical circuit elements on the silicon surface. It is, of course, much more cost-effective, as well as easier from the engineering standpoint, to process large surfaces of silicon rather than to process one chip at a time. A typical silicon chip might have dimensions of 1 cm. by 1 cm., while a typical *wafer* of silicon, as it is called, might be a slice from a cylinder of crystalline silicon with a diameter of about 15 cm. Thus a single wafer would have enough area on it to accommodate on the order of 100 chips.

Not all of these chips, however, are likely to be functional. Various problems, such as defects in the crystalline structure of the silicon wafer, or errors in alignment of the masks, cause a large percentage of the chips to fail at the testing stage. Typical figures for net yield are as low as 10% for large circuits [Ullm84a]. As a first approximation, this figure for yield falls at an exponential rate for a constant factor increase in the area. For example, if area increases by a factor of $c > 1$, the probability of obtaining an unflawed chip falls to $(0.1)^c$. This motivates a major concern in the design of large VLSI chips, which is to reduce the total area occupied by a circuit. A reduction in the total area for a given task would also permit a designer to put more circuitry on a single chip, leading to more functional power available on a single chip.

Chip area is not the only factor of interest in the design of circuits. Speed of operation is another property of vital interest, as are power consumption, heat dissipation, etc. All these factors are interrelated, and for a given circuit design, with all device sizes determined, bringing down the total area will result in bringing down the chip delay. As a first approximation, area can be considered to be the factor to be optimized [Ullm84a].

Consider the problem of laying out a VLSI circuit on a chip so as to minimize its area. This problem has been well researched, and broken down into independent problems of circuit placement and wire routing. Solutions for these problems and their own subproblems have been attempted for over two decades. However, owing to the fact that many subproblems are NP-hard [Sahn80a, Dona80a], progress has been slow.

During the early stages of the development of circuit layout algorithms, it was usually the case that automatic circuit layout packages fared poorly in comparison with manual placement [Prea86a]. This monopoly of the VLSI layout domain by manual designers has given way to design automation: computer-aided design (or *CAD*) programs now do the bulk of the placement and routing tasks.

There are two factors behind this trend away from manual design. First, circuit layout algorithms have grown increasingly sophisticated. Coupled with the greater power of modern-day computers, this provides a quality of automatic layout comparable to or better than that of manual layout. Second, the number of devices that can be placed on a single chip has grown to the extent that manual circuit layout is no longer feasible in a reasonable time-frame, whereas CAD programs can finish the job much faster. Although the CAD programs are computationally expensive, the lead time for a CAD program is far less than the expected lead time for manual layout. Under these changed conditions, the common scenario is for CAD programs to undertake the initial placement and routing jobs. Human designers provide the final scan, hand-packing the layout to squeeze the last bit of additional improvement that might be overlooked by a program.

At this point, one question to be answered is: at what level of the layout process should research into design automation be aimed? One possible area of research is silicon compilation: the completely automatic synthesis of a circuit given its functional description. A silicon compiler would be responsible for *all* aspects of circuit design, from determining the sizes and interconnections between active circuit elements to the final layout of the circuit. However, during silicon compilation, placement and routing stages would have to be included. Hence, the silicon compiler needs to use a placement and routing algorithm in order to generate its circuit layout. As yet, there are no algorithms with any proven bounds on the output, and there is still much scope for new algorithms. Moreover, there are many well understood and thoroughly analyzed designs for building-block circuits, or modules, in VLSI design. Basic circuits such as PLAs, ROMs, RAMs, multiplexors, decoders, etc. have been designed, tested, and packed into very compact layouts. These circuits have been in existence for a number of years [Mead80a]. An approach to placement which begins with these basic circuits, carrying out the job of placing them and connecting them together correctly, would be applicable in its own right as a tool for placement of a pre-designed logical circuit, as well as being available as a stage of a silicon compiler.

Before we proceed further we need to standardize some terminology. We speak of a *layout* as a realization of a circuit on a single chip of silicon in some technology such as nMOS or CMOS. A *module* is a circuit laid out within a bounding rectangle. Typically, modules are subcircuits combined to form the entire chip, each module taking care of a functionally independent task. Connections between a module and the outside world go through *terminals*, each terminal being assigned a position on the bounding rectangle of the module. A *net* is a wire connecting two or more modules, represented by a number assigned to each net. A *channel* is the area between two modules that is reserved for routing the nets. Channels may be concave polygons, or they may be decomposed into rectangles.

The automation of VLSI circuit design can be divided into three subproblems: *initial placement*, *global routing* and *channel routing*. The placement problem is to find a position for each module such that certain properties are optimized. These properties include total chip area, total interconnection length, crossing count (the number of unavoidable crossovers) and maximum number of nets assigned to any channel. Placement is often not an independent phase of the overall design process; rather, it is one step in an iterative process, where a fresh placement may be requested because of insufficient space being allotted for a channel.

The problem of packing a set of rectangles into a bounding rectangle so as to minimize the total area is NP-complete, being reducible to the problem of two-dimensional bin-packing [Gare79a]. The placement problem has been shown to be NP-hard [Sahn80a, Dona80a], and there are no known sub-optimal approximation algorithms for this problem. Recent research in complexity theory gives us no reason to expect a quick resolution of the P = NP question. Moreover, the problems typically encountered in circuit design projects are of such a size that exact solution by an exponential-time algorithm carrying out an exhaustive search is not reasonable. This situation makes heuristic algorithms the only alternative.

The global routing problem follows from the placement: once a relative positioning has been decided upon between the modules, how should the nets be assigned to channels? In other words, what path should each net take in going from one module to another? There could be several choices, and once again the quantities to be minimized are total interconnection length, crossing count and channel area. This problem is also NP-complete, as shown by Pinter [Pint83a].

The channel routing step is the moment of truth for the layout program, since it is usually at this stage that a proposed placement and global routing are either found acceptable, or are found unworkable and rejected. The channel router may find that the estimated channel area between two modules was insufficient. This may be either due to the placement algorithm putting two modules too close, or because the global router routed too many wires through certain channels. There are many ways in which layout schemes handle these problems: for instance, the global routing phase could be repeated to distribute the wiring more evenly across the available channels. Additional tracks could be introduced in the channel area, as is the case with TimberWolf [Sech86a]. The PI package, on the other hand, repeats the whole process starting from the placement phase [Rive82a], as do other proposed placement algorithms [Goto79a, Laut79a]. There is, however, no rigorous method for dealing with these problems, and certainly no known theoretical foundation for choosing one over another. Heuristic rules form the basis for the existing algorithms.

In the area of channel routing, however, there has been a significant amount of work, and heuristic algorithms are now available to do detailed channel routing with nearly optimal results [Reed85a]. Channel routing with *doglegging* is a variation in which the wire is allowed to change tracks arbitrarily. Although this technique has resulted in many related algorithms with very good average-case behavior [Deut76a, Yosh82a], the dogleg channel routing problem has also been proven to belong to the class of NP-complete problems [Szym85a]. Although there is no existing proof of sub-optimality for any of these algorithms, several difficult problems in detailed channel routing [Deut76a] have been successfully routed using only a negligible amount of additional routing area. For example, Sechen and Sangiovanni-Vincentelli [Sech86a] claim that the YACR2 channel router consistently routes channels within 1 track of the channel density.

Although channel routing is an important issue, we have chosen to concentrate on placement and global routing; and, in particular, on the interaction between these two problems. Our approach begins with a placement algorithm that tries to combine two different approaches to partitioning the set of modules prior to placing them. We then propose a novel algorithm to do global routing incrementally during placement, rather than as a separate phase. We believe that the placement phase can provide an insight into the way global routing should proceed, and that this "insider information" is lost when global routing is a separate phase.

This thesis is organized in the following manner. Chapter 2 is a survey of some of the popular algorithms for placement, with examples to illustrate the approaches used by these algorithms. We also briefly preview our approach to placement in chapter 2, contrasting our approach with the other approaches surveyed. We argue that the existing algorithms have overlooked the question of why humans have been more successful than algorithms in this area, at least for small to moderate-sized layouts. Chapter 3 surveys some global routing algorithms, with two case studies of layout packages that have been designed and implemented. Chapter 4 presents the overall layout algorithm, including both placement as well as routing. Beginning with a description of the graphs used to represent the layout picture abstractly, it then proceeds to present the placement algorithm in detail. Global routing and channel routing issues are discussed in chapter 5. Finally, we close with a discussion of the results obtained in chapter 6, and present a summary of the work and future directions for research in chapter 7.

Chapter 2

INITIAL PLACEMENT

The placement phase is critical to the global routing and channel routing stages in the circuit design process. A good placement will allow wire-routing to be successfully completed without much need for rerouting the wires. On the other hand, if the placement is not very good, it might result in some channels being used for a large number of nets (i.e., overloaded), while other channels might bear less than their fair share of nets. In such a situation, many iterations may be needed in order to distribute the wires more equitably across the channels. Overloaded channels need to be avoided because if the channel size estimates of the placement phase are significantly off the mark, then some channels may need to be drastically resized, causing the final layout to have a different shape from that obtained by the placement phase.

The definition of the placement problem given in the introduction is a simplified version of a much more complex problem. In its generalized form, the problem could include modules with arbitrary shapes, non-Manhattan (i.e., non-rectilinear) geometries, overlap of modules, etc. We restrict our attention to the restricted placement problem as defined, i.e., a set of on-overlapping rectangular modules with terminals of nets on the edges, to be placed in such a way that it minimizes an evaluation function.

The placement phase may itself be broken into two steps, depending on the algorithm used to do the placement. The first step is *constructive placement*, when the system first looks at the set of modules and makes some crucial decisions, such as:

- In what order should the modules be placed?
- Which modules should be placed close to each other?
- How should the modules be oriented relative to the neighboring modules?

Following initial placement, the next step is to attempt improvements to the layout, i.e., the *iterative placement* step. Some of the more popular quantities used for measuring improvements are [Leis80a]:

- Total chip area: the sum of the individual module areas and the channel areas. Chip area is the major quantity that we would like to reduce for the obvious reasons that increased area leads to decreased chip yield. Moreover, assuming that the problem is of module placement (i.e., device sizes are fixed), increased area in general means longer wires, leading to higher signal-transmission delays because of the increased capacitance of the wires [Mead80a].
- Total edge length: the sum of the lengths of all the wires interconnecting the modules. The price paid by high edge lengths is the increased chip delay, as described above.
- Crossing count: the number of wire crossings. Each crossing eliminated amounts to an easier job of routing at a later stage, and hence to a greater likelihood of acceptance of the proposed placement.

- Maximum channel density [Souk81a]: Channel density is the maximum number of wires at any point running in parallel along the channel. High density in a channel means that the channel needs to be wide to provide enough space for the wires. This in turn affects the placement, since the algorithm needs to take note of the wider channel requirement and adjust the positioning of neighboring modules. The maximum channel density, therefore, is a good target for reduction, although this implies that we also need to worry about global routing while doing the placement. Currently, Loosemore's COMPEDA system is the only widely known algorithm that combines the placement and global routing processes [Loos79a].

2.1. Constructive Placement

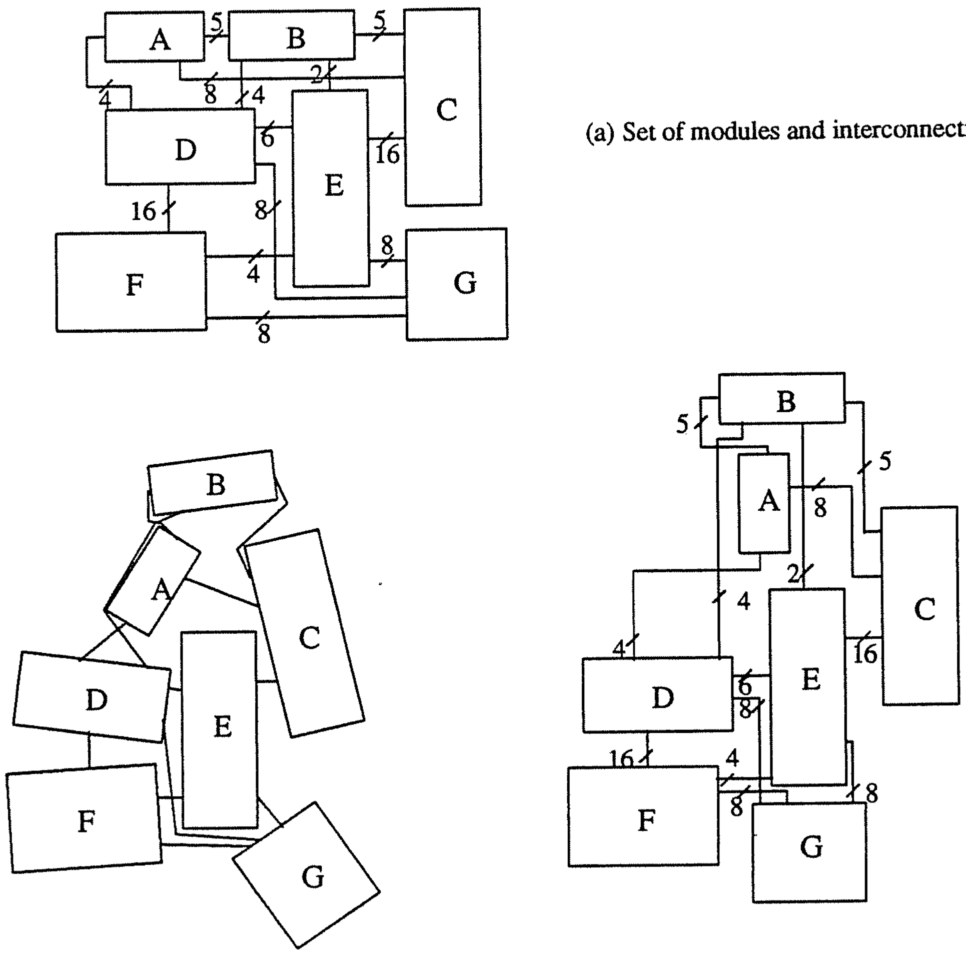
The first stage in producing a placement is constructing an initial placement. The input to this stage is a list of modules, each with its dimensions and the lists of nets on each of its sides. Some approaches use additional information, such as the relative importance of some nets, which may be provided as weights on the nets. Also, while some algorithms are designed to run independently, others interact with the circuit designer to varying extents. For example, Preas and Gwyn [Prea78a] implemented three levels of interactive capability in their placement program: completely automatic, automatic but influenced by user hints, and semi-automatic with a partial placement provided by the user. Our interest lies mainly in fully automatic algorithms, and hence we restrict our attention to those algorithms which are either totally independent, or have a mode of functioning that is independent of user interaction. We further classify fully automatic placement algorithms into four broad categories [Prea86a]: *global approach, branch and bound, top-down partitioning based,* and *bottom-up cluster-growth based.*

2.1.1. The Global placement approach

The first class of constructive placement algorithms that we consider differs from the other approaches in one crucial respect: any placement decision affects *all* modules at the same time, rather than some subset. The global placement approach includes such algorithms as Quadratic Assignment [Hana72a] and Force-directed placement [Quin75a] Although not currently in use in their original form, global placement algorithms have been combined with other approaches with a reasonable degree of success [Wipf82a, Blan84a].

Force-directed placement considers the placement problem to be analogous to the following static physics problem: given a set of blocks on a frictionless plane attached to each other by a network of springs, with a given tension in each spring, what is the equilibrium state of this system? This statics problem is solved by using a system of first-order differential equations to represent the initial state of the system. Figure 2.1 shows the steps involved in a force-directed layout.

In this analogy, each block represents a module, each spring a set of nets, and the tension in a spring is proportional to the number of nets between the two modules involved. A closer model would actually include as many springs as the number of nets, with distinct attachment points, instead of representing a set of nets as a single spring of some tensile strength. With this modification, we can take into account the positions of the nets on the module boundary. This would force rotation of modules to a preferred orientation.



(a) Set of modules and interconnections

(b) Blocks-&-springs equilibrium state

(c) Approximate Manhattan orientations

Figure 2.1: Force-directed placement

Beginning with the set of blocks and interconnections as shown in the schematic diagram of the example layout of Fig. 2.1(a), the algorithm computes the equilibrium position of the system of blocks and springs. One possible equilibrium position is shown in Fig. 2.1(b). This equilibrium position then needs to be adjusted in order to force the modules to conform to the Manhattan layout constraints of the technology, as shown in Fig. 2.1(c). Note that the initial layout of Fig. 2.1(a) is only for our visualization, and is not part of the input to the force-directed placement algorithm; the only information provided to the algorithm is the set of blocks and their interconnections, not their relative positions.

The force-directed approach seems to be reasonable, and it would appear that it should produce a good initial placement. There is one major problem with this approach: *multipoint nets* cannot easily be modeled [Souk81a]. Since a multipoint net would have to be modeled by a spring with a Y-connection, the branch points on the Steiner tree would have to be decided in advance (i.e., we would have to decide where the Y-connection is to be made), and this information is not known at problem specification time. A related problem arises with *multiport nets*, where there is a choice of two or more ports on a module boundary corresponding to the same net. In this situation, the most convenient port should be chosen for connection to other modules. However, the blocks-and-springs analogy does not provide any facilities for automatically choosing between a pair of possible positions to attach springs. Hence, the statics problem would have to be formulated by arbitrarily selecting one port out of the set of available ports. This would lose whatever flexibility might have been gained by having the multiport net in the first place.

Force-directed placement has another drawback: it does not consider channels and wiring space during its placement phase. Channel sizing is a separate phase that is executed after the placement is completed; and after allocating space for the channels, it is possible for the final placement to have little similarity to the initial placement and its apparent compactness.

Another algorithm falling under the class of global approaches is *convex function optimization*. This algorithm constructs an objective function that depends either on the number of interconnections and the distances between pairs of modules, as in the quadratic assignment formulation [Hana72a] or on some convex metric, such as a least squares metric, as in a convex function optimization formulation of the problem [Blan84a]. In the quadratic assignment formulation, for instance, the objective function to be minimized might be defined as:

$G = \Sigma\, c_{ij} \times d_{p(j)p(i)}$, where

$C = [c_{ij}]$ is the cost matrix, representing the cost of the connections between modules i and j;

$D = [d_{kl}]$ is a distance matrix representing the distance between positions k and l; and

p(i) represents a permutation of the components' positions.

The objective function is then minimized over all permutations of the component positions.

The drawbacks of these optimization metrics are common ones: they are unable to deal with practical constraints, such as finite areas for the cells, fixed positions for some of the cells, and the necessity to impose Manhattan orientations on the final form of the layout [Souk81a, Prea86a]. Moreover, in the case of the quadratic assignment formulation, there are additional constraints: only two-point nets can be modeled; and an optimum solution to the quadratic assignment problem does not necessarily imply an optimum solution for the associated placement problem.

2.1.2. Branch and bound

The branch and bound algorithm has been used in a wide range of applications, typically in situations where an exhaustive search of the solution space appears to be unavoidable [Hill80a, Lawl66a]. Heuristic solutions to many NP-complete problems are obtained by this technique, such as the Traveling Salesman problem, the Hamiltonian Path problem, etc. Although it is computationally expensive, there are often no acceptable alternatives. The method is named after its behavior of generating a tree of all possible partial solutions and discarding branches of the tree by using some known bound lower than the estimated cost of the discarded subtrees.

In the context of the placement problem, the branch-and-bound algorithm starts by picking some module, and then generating one subtree for each possible position of that module in the placement [Hana72b]. In each subtree, it uses a heuristic cost function to determine the minimum cost of that subtree; in other words, it tries to predict the best possible placement that could be obtained within that subtree, assuming a fixed position for the chosen module. Once this cost function has been computed for each subtree in the tree, the costs are compared with some known lower bound, perhaps from a random placement that is first carried out simply to provide a bound. All subtrees with an estimated cost greater than the known lower bound are abandoned — obviously, it is pointless to work any further on a placement if, with just a few modules placed, it costs more than a known placement.

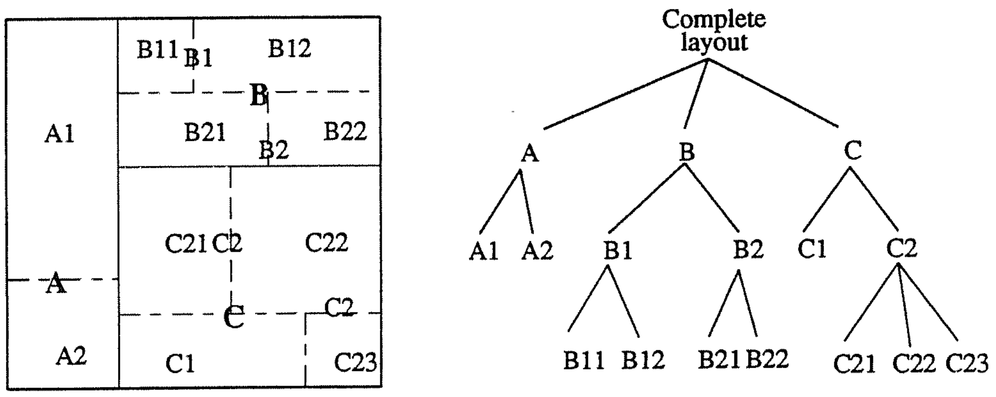
Having thus removed the more expensive branches from the tree, the remaining subtrees are further expanded by choosing another module and generating one new subtree for each position of the next module chosen, with the first module's position fixed. As the tree gets deeper, more modules are placed and more subtrees are eliminated because of their high cost; thus the algorithm avoids the problems of an exponential increase in the number of layouts to be considered.

Although branch-and-bound is an elegant and powerful technique in general, its use in placement is limited by its high computational complexity. In order to reduce the set of subtrees to tractable limits, a tree-pruning heuristic function needs to be formulated; however, it has also been noted that the more accurate the heuristic cost function, the greater its computational complexity [Prea86a]. As a result, there appears to be no way out of a high computational cost associated with this approach. Moreover, the branch-and-bound technique can significantly reduce the problem size only if the early subtrees close to the root can be pruned; this, of course, can never be guaranteed. The increased size of modern-day circuits aggravates this problem to the extent that branch-and-bound is not in popular use as a placement algorithm.

2.1.3. Top-down partitioning

We now turn to the top-down approach to partitioning the set of modules into subsets for placement. Algorithms based on partitioning the module-set into subsets are widely used in modern layout systems. The rationale behind the top-down partitioning process is to reduce the number of wires in the center area of the layout by partitioning the module set across a small number of interconnections, thus moving the highly-connected modules away from the center. Among the available top-down partitioning algorithms, two approaches merit a detailed description, based on their popularity: ***hierarchical placement*** and ***min-cut placement***.

Hierarchical placement is an approach that depends on the user to provide the chip hierarchy. The initial information provided to this algorithm includes some version of a "decomposition tree", describing the hierarchy of the components [Prea78a, Wipf82a]. For example, as in Fig. 2.2a, the designer may have designed



(a) Partition of layout area (b) Hierarchical decomposition tree

Figure 2.2: Hierarchical layout example

modules A1 and A2 as subcircuits of the enclosing module A; likewise with modules B and C. This hierarchical decomposition may be several levels deep.

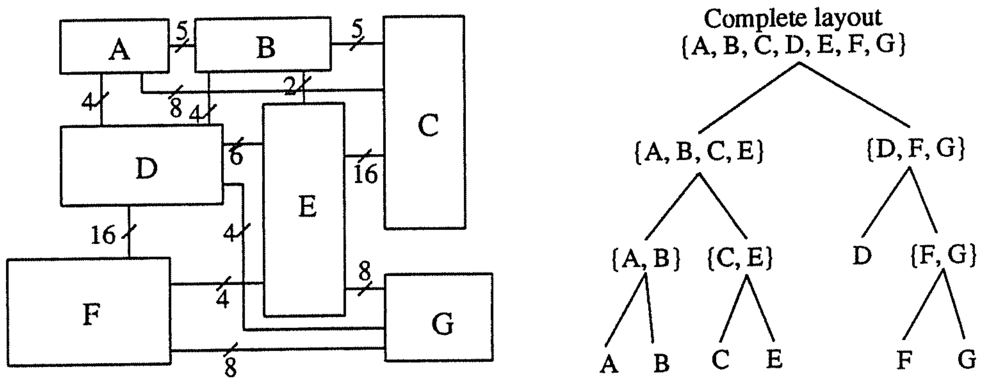
The tree representation of Fig. 2.2b shows how the chip might be partitioned into subproblems. Leaf-nodes correspond to the smallest (atomic) modules, which cannot be partitioned any further. An internal node is a module whose components are the smaller modules living in the subtree of the internal node. The subsets in each subtree of an internal node X reflect the structure of the chip, since modules in each subtree are more intimately related to each other than to modules in other subtrees of X.

Placement is done on the lowest level subtrees first, in the manner of a recursive postorder tree traversal, bottom-up. Each subtree is treated as a separate problem, and placing modules in a subtree ends with the entire subtree built into a new module, i.e., a rectangle with terminals on the periphery. As a subtree is processed, it is replaced by the new module containing the modules of the subtree.

A major drawback of hierarchical placement is that sometimes VLSI layouts may not have a strict hierarchy enforced during the design. As a result, the decomposition of the hierarchy may not be isomorphic to a tree, as might be the case when some area of a layout mask is realized by overlapping two modules. Moreover, the placement algorithm is dependent on user-interaction to provide the decomposition tree. Hierarchical placement has often been used in layouts containing only a few modules. But with the size of layouts that designers are faced with today, it is very difficult for a designer to assimilate and analyze the design to the extent of being able to provide meaningful information. Although designers do provide valuable information, their perception of the circuit may bias the hints they provide, overlooking some possible choices in the layout design decisions. In order to provide the ability of investigating such possibilities, a reasonable design for a placement algorithm would allow human designers to provide hints for the algorithm to use or discard at its discretion. Algorithms may then be able to uncover good placements that the human designer might have overlooked, while paying heed to the wisdom of the designer.

Ideally, the placement algorithm should provide additional information to augment the user's intuition, in the form of suggested groupings of modules. The min-cut family of algorithms does, in fact, carry out placement with such information in mind. Most recent research in placement algorithms has included minimum-cut placement as a part of the solution [Rive82a, Wipf82a, Laut79a]. The minimum-cut approach partitions the set of modules into subsets and recursively lays out each subset, much in the manner of the hierarchical placement technique. The difference is in the way the partition is determined. A minimum-cut algorithm is used to break up a graph into two subgraphs such that each subgraph has about the same total area of modules, and such that the number of edges between the subgraphs is minimized. Thus, minimizing interconnections between subgraphs increases the interconnections *within* a subgraph.

The number of nodes in each subgraph may be bounded from below; for example, each subgraph may be constrained to contain at least one-third of the nodes in the original graph. Such a partition would ensure that the decomposition of the chip into sets of modules would not result in trivial or degenerate trees, i.e., it would ensure that each node would have subtrees of about the same size. Both routing area and total wiring length should be decreased by following this heuristic.



(a) Module set and connections (b) Min-cut partition tree

Figure 2.3: Minimum-cut placement

As an example, consider Fig. 2.3a, where a number on the edge indicates the number of nets represented by the edge. What is the minimum-cut for this problem? The decomposition tree shown in Fig. 2.3b shows the minimum-cuts at each stage, ending with the leaf nodes; at the topmost level, this partitions the modules into two sets {A, B, C, E} and {D, F, G} with 26 edges between them.

The biggest shortcoming of the min-cut approach to module placement is the difficulty of determining the min-cut. In general, if the size of each subgraph in the minimum-cut partition is bounded, then the problem is NP-complete [Gare79a]. Thus, some heuristic is needed to compute a minimum cut, and these heuristics tend to be computationally expensive [Mura80a].

Besides this problem, the bipartitioning approach is often not able to obtain a min-cut which matches a natural partition of the circuit [Prea86a]. Connectivity information (the number of nets between modules) may often contain unsuspected hints for a good layout. A layout tends to be biased by the designer's perception of the flow of control between modules. It is quite possible that there may be situations in which a partition might be counter-intuitive, but result in a better placement. In such cases, minimum-cut placement might reveal surprising ways to partition the chip. On the other hand, situations may arise in which the min-cut partition of the module set is at odds with the flow of control without any significant advantage gained by deviating from the natural partition. For instance, consider a set modules realizing a piece of sequential logic, with the modules connected in sequence. Since min-cut does not use know anything about the function of the circuit, it could place these modules in different subtrees of the min-cut partition tree, if they were lightly connected.

In such a situation, it would be desirable for the min-cut algorithm to choose a partition that does not separate sequentially interdependent modules by a large distance on the chip. Such constraints are not part of the min-cut approach; nor is it apparent that constraints of this form can be added to the min-cut algorithm. The problem is that the additional constraint needs to achieve its results without seriously obstructing the general purpose of the min-cut approach, which is to provide a good layout for global routing. For instance, the heuristic Dyjkstra min-cut algorithm [Dyjk78], which is the algorithm used by the PI system, does not accept such constraints.

2.1.4. Bottom-up cluster-growth based placement

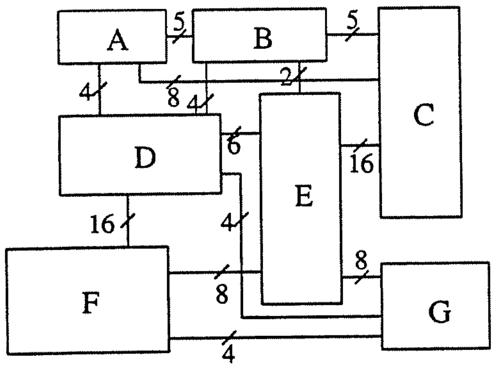
Cluster-growth based placement algorithms have also been in vogue in the past few years. The viewpoint of this approach is similar to that of minimum-cut. The difference is that while minimum-cut takes a top-down approach, clustering takes a bottom-up approach; it sets out to place modules close together if they are highly connected [Rive82a]. Hence it typically starts by picking a module, based on such properties as its size, or the number of terminals, or the number of connections to the input/output pads on the chip, etc. Once a module has been chosen, the sequence of subsequently placed modules is determined by the number of nets going from the unplaced modules to the already-placed ones. Again, there is room for variation in the criteria for selection of the next module: the basis could be, for instance, how thickly the module is connected to some particular module in the placed set, or how thickly it is connected to the entire currently placed set.

The biggest advantage of using clustering is that connectivity information is much easier to compute and manage than minimum-cut heuristics, and hence this approach is very simple to implement, as well as computationally inexpensive. Most algorithms using clustering tend to have a time complexity of $O(n^2)$ [Kurt65a]. Typically, the kind of information required is readily computable, such as the degree of interconnection of the modules in the circuit. Also, as opposed to minimum-cut heuristics, the cluster-based approach does not care about decomposing the module set into equal sized subsets; its bottom-up mode of functioning groups the modules into natural clusters. However, this approach does not provide much help to the global routing phase. With a min-cut partition, global routing algorithms can assume that there will be few nets in the center area of the layout. Although clustering puts highly connected modules together, it may lead to a large number of nets being routed near the center, leading to congested channels in that area and relatively empty channels near the periphery. As a result, global routing algorithms associated with cluster-based placement may need to be modified to take this into account.

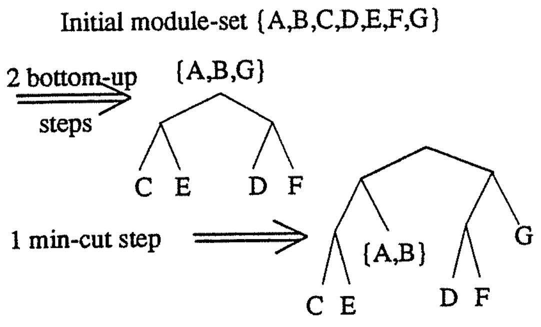
Modern systems tend to use either the cluster-based or the bipartitioning approach to placement. These two approaches have fewer basic problems than either the global placement algorithms or the branch-and-bound approach. The global approach is not able to satisfactorily handle such practical considerations as multipoint nets, multiport nets and wiring space. Branch and bound suffers from high computational complexity, which makes it infeasible for current problem sizes. The hierarchical approach, although computationally feasible as well as practically workable, requires user-interaction and user-intuition, which may not be very dependable for very large problem sizes. Overall, the only approaches that combine workable principles with independence from human intuition are bipartitioning and clustering approaches.

2.1.5. Combined approaches

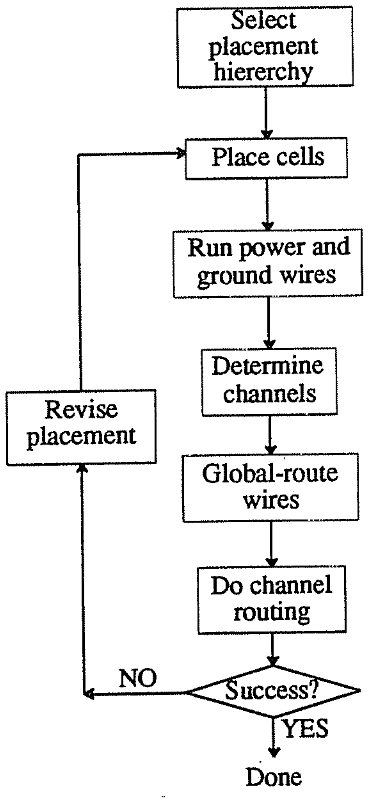
In addition to the algorithms discussed in the last section, combinations of approaches have also been implemented, and some have attained reasonable success. The MIT PI (Placement and Interconnect) project [Rive82a] is an experimental package which merits a detailed case-study, being a good example of combining two approaches to placement. The PI placement's partitioning heuristic behavior is shown in Fig. 2.4(b), along with its iterative design cycle in Fig. 2.4(c). The layout proceeds



(a) Module set and interconnections



(b) Placement steps for example



(c) Iterative design cycle

Figure 2.4: MIT PI placement process

as follows. Based on some evaluation function, a decision is made at each step in the placement process as to whether to apply a min-cut partition or a bottom-up connectivity-based step to the module set. A min-cut step would partition the set of unplaced modules by applying the min-cut heuristic, whereas a bottom-up step combines two closely connected modules into one subtree. Repeated application of this process decomposes the entire module set into a binary tree, with each level in the tree representing one step in the placement process. Each subtree is then placed during a recursive postorder traversal of the partition tree. Channel spacing is approximately estimated and left for later confirmation. Global routing is performed next, and finally the "greedy" channel router [Rive82b] does the detailed channel routing on each channel. If any channel is undersized or excessively oversized, then it reports a failure, and the placement is revised by enlarging or shrinking the offending channel(s). In his paper describing the PI project, Rivest claims that only a few iterations were typically necessary to lay out and route the test examples.

Other combined approaches include a combined force-directed and min-cut algorithm [Wipf82a], a hierarchical approach combined with connectivity-based heuristics [Prea78a], and a placement strategy for standard cells (with some uniform dimension) using a combined bipartitioning and cluster-based algorithm [Rich84a]. Having already discussed the basic ideas inherent in these approaches, we proceed to the placement improvement algorithms.

2.2. Iterative placement improvement

Since initial placement algorithms often produce layouts that can still be improved, most placement packages go through an iterative improvement phase after obtaining an initial placement. In this phase, the placement is repeatedly perturbed by making local changes; the resulting placements are then evaluated. The changes may involve the position of a module, its orientation, or the relative positions of a group of modules. There may be several iterations before a final placement is accepted, resulting in a final layout that is very different from the initial placement. The modifications are accepted if the resulting placement is superior to the original placement.

Pairwise exchange is the simplest of the improvement schemes. This algorithm repeatedly picks a module, and some k modules in its neighborhood. Within this set of modules, there are $k \times (k-1)/2$ possible pairs of modules. Each such pair is exchanged, and the exchange resulting in a placement of least cost is determined. The best of these placements is used as the starting point for another pairwise exchange centered on another module.

Naturally, this is an expensive exercise. Typically, a design has several "core" modules (i.e., modules considered important enough to serve as centers for starting a pairwise exchange pass) and each module may be surrounded by 7 or 8 neighboring modules. For each core module, the entire placement must be re-evaluated some 30 times to produce an iterative improvement. Moreover, the exchanges will be difficult to carry out if the modules are of diverse sizes. In such a situation, an exchange would have to be accompanied by a re-computation of positions and channels. With standard cell types of layouts, where all modules are of about the same size, exchanges do not force additional changes in the entire placement. General layouts with varying-sized modules will raise the cost of this procedure, making this approach computationally expensive. In spite of these problems, pairwise exchange is a widely used iterative improvement algorithm.

A more sophisticated version of the pairwise exchange scheme involves more modules in the improvement iteration. Instead of attempting to do $k \times (k-1)/2$ pairwise exchanges, all k! possible permutations within a neighborhood of k modules could be tried. If the layout is a gate-array type of layout, with uniform-sized modules arranged in rows, we might try all 2^k possible exchanges between 2 rows of k modules. These approaches are even more prohibitively time-consuming than the iterative pairwise exchange; naturally, they also produce more improvement than the limited pairwise exchanges could.

The basic problem with the exchange schemes is that such approaches do not have the capability of getting past *local optima*. A local optimum is an optimum value for the placement evaluation function within some limited neighborhood. Since the improvement algorithms discussed only accept changes when the cost is reduced, they locate only local optima. It is often the case, however, that if a large enough perturbation could be introduced, the optimum value in the new neighborhood could be better than the previous local optimum. Unfortunately, large changes are rarely tried, for the simple reasons that they are computationally expensive, and that there are too many possible large changes for the program to consider them all in a methodical way within a reasonable time-frame. The *relaxation* algorithm was developed by Goto [Goto79a] as an attempt to overcome the local optimum problem.

The relaxation algorithm starts with an initial placement produced by an algorithm such as those discussed in section 2.1. Given this initial placement, a module A is chosen (some reasonable size/connectivity criterion can be used). A best position is computed for module A, relative to the all the other modules in the initial placement, and it is placed in that best position, even if this position overlaps another module. If it does overlap another module B, then module B is selected next, and placed in the best position that it could be in. Each time a module is moved, its possible overlap with another module is handled by forcing the overlapped module to move next.

The process is repeated until either the list of modules is exhausted, or (more likely) there is a loop in the sequence of moves; in other words, when the process attempts to move a module which had already been moved. When such a loop is encountered, the modules involved in the loop are all considered, and one is selected as the winner; the winner gets to keep its position, forcing the other modules to iterate out of its way. The relaxation then proceeds with the unplaced modules. During this continued relaxation loop, unplaced modules may overlap with each other, but they may not overlap the winning module of the previous relaxation loop.

The reason relaxation does better than pairwise exchange is that the loop of constraints between modules tends to span many local optima, and gives the algorithm a chance to look at and choose from a range of local optima. In other words, relaxation goes over a larger neighborhood and has a wider choice of configurations. This is quite a bit better than getting stuck in and being unable to extricate the placement from a local optimum, as was the case with pairwise exchange. The iterations are not much more expensive than pairwise exchange, but the method allows a wider range of possible placements to be examined.

Relaxation is most conveniently applied to problems dealing with modules of approximately the same size, for reasons already stated. In Goto's paper, for instance, all cells were of the same (or almost the same) size, and the "best positions" computed were all made to coincide exactly with the position of the overlapped module; partial overlaps and multiple overlaps were not considered. Intuitively, it is easy to see that different sized modules cause more problems here than in pairwise exchange. Both approaches share the problem of modifying the entire placement if varying-sized modules have to be moved. In addition, if modules can be of different sizes, then exchanges may cause the moved module to overlap with not one but several other modules. This leads to a blowup in the number of modules that are forced to relocate. Hence relaxation may fare better than pairwise exchange in the uniform-size model, but it behaves very poorly when faced with nonuniform-size model, and is best suited to gate-array or standard-cell types of layout problems.

Simulated annealing is a new technique that is based on the physical process of annealing used in the manufacture of glass and ceramic plates, metal sheets etc. Annealing is the process of cooling a substance at a slow and controlled rate, so that the material cools evenly. This allows all internal stresses to even out, with the result that the material settles down in a low-energy state, and should therefore be less brittle and more resilient. In the circuit placement analogy, the cost of the placement corresponds to the energy of the state; thus, simulated annealing attempts to find a "low-energy" placement, i.e., a low-cost placement.

The key idea in annealing is that by providing enough heat to slow down the cooling rate, enough energy is also provided for the material to *gain* energy to a limited extent, which allows it to settle down in the more stable of the possible states. Figure 2.5 shows an example of what the "energy levels" might be for a set of different placements. Viewing this graph as an analogy of the energy levels of a sheet of

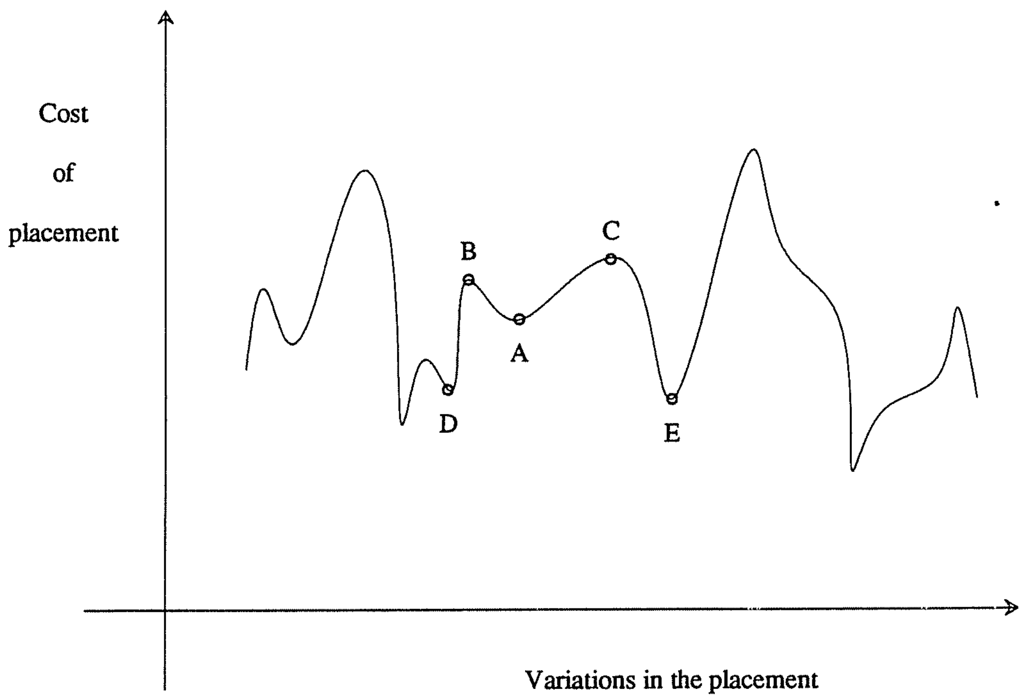


Figure 2.5: Simulated annealing in placement

glass, it can be seen that if the glass is cooled swiftly, it might find itself in the state labeled A in the graph. On the other hand, if the cooling were slow, and enough energy was available, then from state A, the glass might be able to absorb enough energy to reach state B or C; from where the most stable energy level would be D or E, rather than A. The reason why this works is that by allowing the system to temporarily *gain* energy, a wider neighborhood of possible states is sampled in looking for the stablest, lowest-energy state.

The algorithm for simulated annealing can be stated very simply as follows. Perturbations are again made in the layout, where each perturbation is typically a pair-wise interchange of modules. A perturbation is accepted if it results in a lowering of the cost. If the cost increases, then the change is accepted with probability $e^{-\Delta c/T}$, where Δc represents the change in cost due to the perturbation, and T represents the "temperature" of the system. During the simulation, the temperature is reduced by repeatedly multiplying it by a function α, called the *cooling schedule* [Sech86a]. Thus, after a perturbation loop at temperature T, the new temperature is determined by:

$$T_{new} = \alpha(T_{old}) \times T_{old}$$

Simulated annealing has been adopted by several placement algorithms as a viable and moderately successful technique to improve on the initial placement. However, it is also computationally very expensive. Annealing schedules tend to be more successful as more "hill-climbing" is allowed; i.e., as the energy level is allowed to get higher. This can only be achieved by slowing down the cooling schedule, which can only be done at the expense of processing time. An analysis of the comparative performance of simulated annealing against a min-cut algorithm showed, surprisingly, that while simulated annealing was at least 100 times slower than min-cut, it did not produce significantly better layouts than min-cut [Hart86a]. Moreover, in the same paper, the point is made that simulated annealing also involves several arbitrary parameters in the computation of the cost function; in spite of extensive experiments, no consistently optimal set of parameters could be obtained. In conclusion, Hartoog says:

> While we believe Simulated Annealing will continue to be of considerable theoretical interest, it does not appear to be useful in a production environment, even when the highest quality results are desired.

In the light of the iterative approaches discussed above, we need to point out that the PI project goes through a very different style of iteration from the iteration of the simulated annealing or relaxation algorithms. The processing cycle, as discussed in section 2.1.6, is typical of many constructive placement schemes that do not use any iterative improvement as a special phase. Placement improvement has a set of possibilities that it tests out one after the other. According to the PI philosophy, on the other hand, possible placements are tried out based on specific failure information obtained from the channel router. The argument put forward by Rivest [Rive82a] for this omission is that PI's constructive placement produces good layouts, based on the right properties of the circuit; and iterative improvement is not really helpful for such placements.

One result of this philosophy is that the iteration occurs at a time when all the global routing and channel routing has been done. In section 2.1.5, we surveyed the layout style used by the PI algorithm, and we noted that the loop to determine a valid layout encompassed a large number of computationally expensive stages. We now contrast these two different, computationally expensive, iteration styles with the philosophy of our placement algorithm.

2.3. Proposed placement philosophy: an overview

In all the above algorithms, one point of view is consistently missing: how a **human designer** would react to a given problem. In these problems, as in many other similar hard problems, human minds have always shown much better ability at intuitively jumping to almost-optimal or optimal answers. None of the above approaches tries to use this point of view. Each algorithm touches on one aspect of the human approach, but no approach attempts to combine the qualities that make human designers better than most programs. We have made the point that problems have grown beyond the point of employing human designers to find manual solutions. However, it is still instructive to study the approach that a human designer takes.

Let us consider a designer sitting at a CAD terminal during a VLSI design implementation session, in the process of putting down modules on a frame. How does the designer pick the next module to place? How does he (or she) decide where to put it down, and how to connect up the nets? The typical approach of a human designer is to approach the layout problem with a great deal of hierarchical information about the circuit. By the time the layout stage has been reached, the designer knows the functional relationships between the various modules. Hence the designer tends to lay out the modules in roughly the order in which they were designed, i.e., in order of importance of the modules. This can be variously denoted by the order of decreasing number of terminals, or of decreasing connectivity, or of size.

However, this is not an absolute rule. The designer applies this rule to perhaps 2 or 3 levels, then abandons that stream of control flow and picks up a different stream. Thus, there is a point at which any single sequence of dependent modules is abandoned, and another one is started. The leftover modules are usually the unimportant (or peripheral) modules, which are taken care of at the end of the layout process, and fitted in wherever there is space.

Another consideration that is at the forefront of the designer's mind is shape and orientation information. A designer looks at the next module to be placed, and at the existing layout, and visually determines roughly where the module should go. This is something that is of paramount importance in achieving compact layouts; however, most placement algorithms have either ignored this factor altogether, or relegated it to the final stages of constructive placement and iterative placement improvement. One should note, however, that this is not the same question as bin-packing. Area is not the only quantity under consideration; there are connectivity constraints that influence the final placement of a module with respect to another. In addition to low chip area, we would also like the placement algorithm to help the global routing phase.

These, then, are the central ideas in our approach to placement. Our algorithm attempts to place modules in order of their relative importance for a few steps. In order to identify these "more important" modules, a clustering algorithm is developed. Keeping in mind the drawback of the clustering approaches, we give this cluster-formation stage a tendency to identify closely connected modules, while at the same time it tries to minimize the connectivity between clusters. Thus we hope that the final layouts will not contain congested channels in the central area of the layout.

The algorithm repeatedly places modules beside the "most important" module of a cluster, using orientation and size information to select the next module to be placed. As modules are placed, they are combined into *super-modules*. Choosing and combining modules into super-modules is also influenced by size and shape information. After reaching a point at which the selected modules are below some "importance threshold", the algorithm abandons them temporarily, leaving them for later placement. The detailed algorithms implementing this approach are presented and discussed in chapter 4.

Chapter 3

GLOBAL ROUTING

Having run through a range of placement algorithms, we now switch to global routing as the next phase of an automatic VLSI circuit layout process. Global routing is the process of determining the topological route that should be taken by each net in order to connect together terminals with the same label. The global router assumes that the placement is done, or at least roughly done, in the sense that all modules are assumed to have known positions relative to each other. It then computes which route each net takes. This computation involves two stages: ***channel definition***, and ***loose routing***.

Channel definition is the stage that identifies the channels through which the nets will have to pass. The ***channel area*** is the empty space between modules; we can either design clever algorithms to handle irregularly-shaped channels, or else we need to break up the channel area into rectangles. Channel definition is not really a very difficult problem and there are several acceptable solutions to choose from [Souk81a, Rive82a, Ullm84a].

Loose routing determines the sequence of channels traversed by each net, given the set of channels. The problem is in picking routes for nets in such a way that the resulting routing is a close-to-optimal routing, in the sense of the measures discussed in the early parts of chapter 2: total area, total edge length, crossing count and maximum channel density. In this phase, the total area is not as sensitive to the global routing produced as the edge length, crossing count, or maximum channel density. The most important quantities should be the total edge length and maximum channel density, since the first influences the overall chip delay, while the routability of the channels depends on the latter. The loose routing problem has been shown to be NP-complete [Pint83a, Sahn80a]. Moreover, it is hard to concoct a function that might represent and measure the "goodness" of a layout so as to reflect all the issues involved. This is because the quantities that need to be optimized are drawn from a set of widely varying and disparate entities. Referring to chapter 2, some of the quantities involved are total edge length, crossing count and maximum channel density: three quantities that have different units, and can't be combined into one single optimization criterion in any rigorous sense. Even in manually laid-out and routed chips, we have no way of choosing one layout over another when the two have very close values for these measures; we do know when one measure is outrageously violated in order to obtain a good value for another. To this extent, a composite function could be built based on heuristics, such that it would reject layouts below a certain threshold for any single measure. This does not, however, lead to any reasonable cost function for optimization. As with placement, so with global routing: heuristic algorithms are needed.

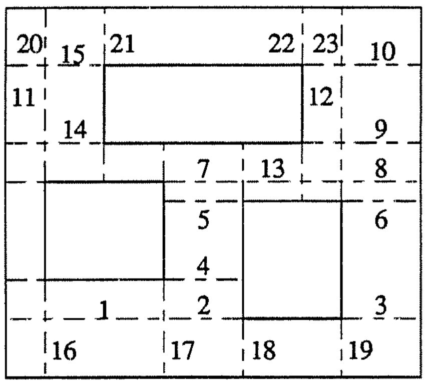
Although much research has gone into this area, the results are not very promising. The majority of the algorithms are modifications of the basic Lee-Moore expansion algorithm [Lee61a, Moor59a]. A second class of algorithms consists of a small number of more innovative algorithms that have surfaced in recent times, but there is to date only one attempt, by Loosemore [Loos79a], that approaches what our proposed algorithm does, which is to combine placement and global routing. The problem tackled by Loosemore, however, is that of approximately uniform-sized modules placed in rows and columns, which is a very restricted version of our problem.

This chapter reviews the channel definition problem, and the two classes of global routing algorithms mentioned above. It then conducts two case-studies: a recent modification to the basic Lee-Moore expansion approach, to see the direction modern algorithms have taken; and the MIT PI project's approach, as a different direction in which the expansion idea can be extended.

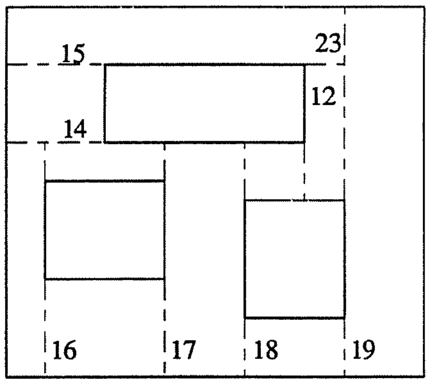
3.1. Channel definition

Consider the simple example of Fig. 3.1, where the rectangles are the laid-out modules. The solid rectangles represent the modules, whose edges are extended as dashed lines. The ordering of channel edges based on the decreasing order of edge lengths of the longer edges have been placed alongside the channel edges. Thus, a channel edge marked 13 is smaller in size than edges marked with lower numbers. A large number of different algorithms all define approximately the same set of channels as the ones shown in Fig. 3.1b. The MIT PI project algorithm is typical of these algorithms. It can be broken down into the following three basic stages:

- Extend all cell edges to meet either the cell or the chip boundary.
- Considering each line segment of the cell edge extensions in the order of longest segment first, delete the segment under consideration unless this would leave a non-rectangular channel area.



(a) Edge-extensions, sorted by length



(b) Unnecessary line-segments deleted

Figure 3.1: Channel definition

- Add border channels if necessary.

In Fig. 3.1(a), a few of the larger line segments have been numbered in order of length, and Fig. 3.1(b) shows the set of channels after deleting all unnecessary line segments in order of size.

There are a few other factors that the channel definition algorithm needs to take care of: avoiding *constraint loops*, and avoiding *switch-box* channels.

A constraint loop is a set of channels such that each is constrained by another in terms of position. The example of Fig. 3.2 shows a constraint loop between channels a, b, c and d: a must lie above b, b to the right of c, c below d and d to the left of a.

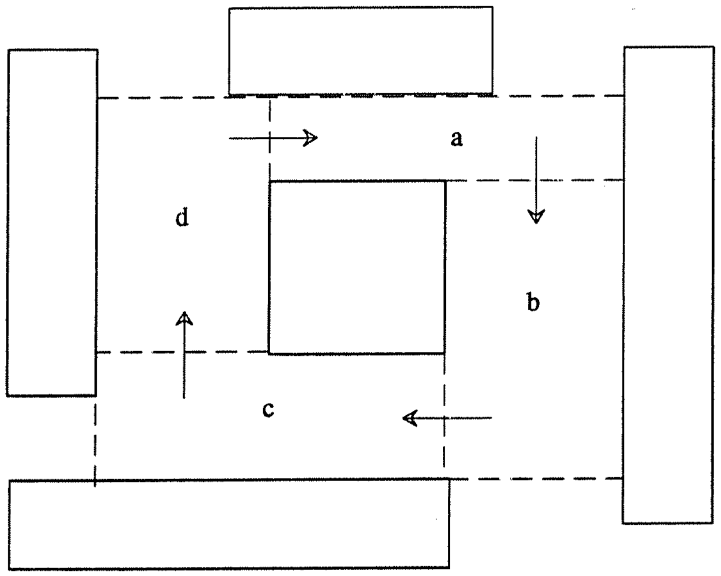


Figure 3.2: Constraint loop between channels

Such a loop could disrupt certain loose routing algorithms, especially if the algorithms depended on information about the nets in adjoining channels while in the process of assigning a net to a channel.

A switch-box channel is a rectangular channel with fixed terminals on all four sides. While many detailed channel routing algorithms have been developed, the switch-box routing problem is still a mostly unsolved one; existing algorithms are unsatisfactory, to say the least, and we would like to avoid this problem if possible.

3.2. Loose routing

Most loose routers in the literature hail from the well-known Lee-Moore algorithm, as it is now termed [Lee61a, Moor59a]. The principle of the algorithm is quite simple: picking a net, the algorithm begins to *expand* the net to all adjacent grid points or rectangles, generating points called *source points*. Each source point is again expanded, and this proceeds outwards from each terminal of the current net, until they meet somewhere. The expansion of source points can be controlled by attaching a cost property to each point. If the cost is made proportional to the distance from the source terminal and the distance to the destination terminal, then at any point during the expansion process, we only need to expand those source points that have the least cost.

This process, when applied to a 2-point net, looks very similar to ripples spreading outwards from a pair of pebbles dropped in still water; the meeting-point of the two expanding circles of ripples will automatically be the shortest distance between the two points. However, when more than one terminal is involved, this needs some modification. As soon as two terminals meet at a point, all points on the established path are given a cost of 0, and the search is continued for the best connection to the remaining terminals.

This algorithm, developed in 1961, is very simple, and it is a measure of the difficulty of the global routing problem that no landmark algorithms have been proposed for this problem since then. Research has concentrated on improving this basic approach.

3.3. Case study I: Clow's algorithm

One recent algorithm proposed by Clow looks at various gains to be made by applying more intelligence to the steps of selecting source points and expanding them during the net-expansion stage of the Lee-Moore class of global routers [Clow84a]. Clow speeds up the selection phase by selecting the next source point based on a complex cost function, and applying artificial intelligence techniques to the potentially explosive number of possible search paths at each step. And he streamlines the expansion process to travel along a channel edge, ignoring any expansion across a channel. This improves the running time immensely, since much of the expansion might go across channels, while we really do not expect to go across a channel until we actually are "across the road", so to speak, from the target terminal.

In Clow's algorithm, the next source point is selected by applying Nilsson's A* algorithm [Nils71a] from AI, which is a tree-search approach with pruning of high-cost subtrees. The algorithm keeps track of two costs for each source point: the cost $g(n)$ of the expanded path from the source terminal to the source point, and projected cost $h(n)$ of the path from source point to destination. The function $h(n)$ needs to be a lower bound on the actual cost; hence the Manhattan distance (horizontal distance + vertical distance) is used for $h(n)$. If the source point already had an associated cost, the algorithm compares this old cost to the computed value of $g(n) + h(n)$, and sets the cost as the lower one (which also determines the lower-cost path from the source terminal to the current point).

Expansion of source points does not need to proceed in units of one grid dimension. We only need to note that if the cost decreases on entering a channel, then it is

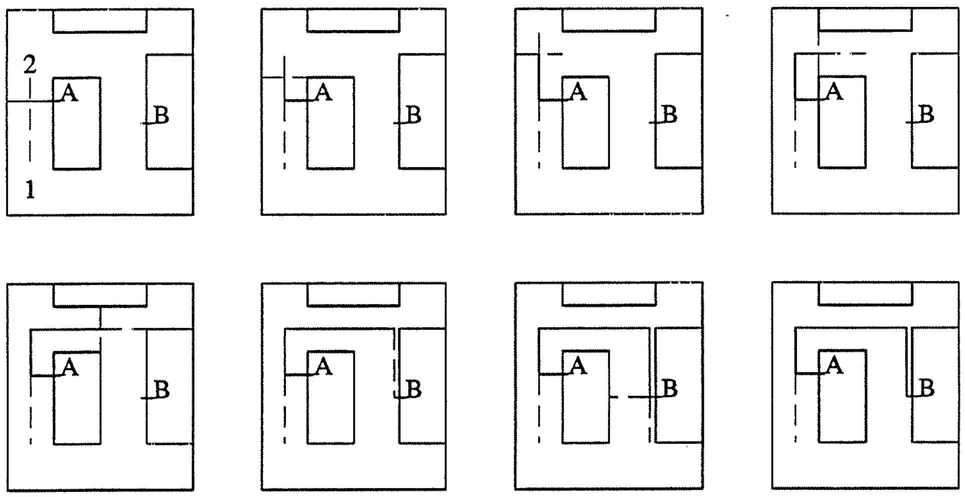


Figure 3.3: Clow's global routing algorithm

most likely that we want the net to traverse the entire channel. Hence, instead of going point-by-point within a channel, the channel expands right to the end of the channel. To take care of other eventualities, the algorithm actually expands along all possible directions, but the next source points are the points of intersection with the channel boundaries. The example of Fig. 3.3 shows, in steps, the expansions tried (in dashed lines) and the successful expansions (in continuous lines). The expansion to point 1 has a larger *g(n)* than that to point 2, forcing the expansion of point 2 first. From then onwards, the upper route never drops below the other in cost, so that it is chosen as the winning route. The channels have not been drawn explicitly for reasons of clarity.

3.4. Case study II: PI project approach

The algorithm used by PI is somewhat more than just a modification of the Lee-Moore algorithm. The expansion process is similar to the Lee-Moore algorithm, but expansion is driven by Dijkstra's single-source shortest path algorithm [Aho83a]. This is a dynamic programming algorithm that finds the shortest path between two points in a graph containing weighted edges. In the channel routing problem, the expansion proceeds by jumping between midpoints of channel boundaries. It is assumed that there are edges connecting any two points between which we wish to travel; the length of an edge is its physical length.

Again, a generalization is necessary when more than 2 terminals are involved. To quote from Ullman [Ullm84a],

> When a net consists of more than two points, PI uses a generalization of Dijkstra's algorithm, where we push out from all ports on the net simultaneously, extending the set of points (midpoints of channel boundaries) reached from each point in such a way that at all times the points below a certain distance have been reached, and points above that distance have not been reached... as the wavefronts of reached points about each port expand, at times there will be a point just reached from one port that borders a channel previously reached from another. If that is the case, we know the channel in which these two paths will join, and we permanently join them, allowing only the wavefront from one of the two ports to continue expanding.

The algorithm finds shortest paths for each net. However, it doesn't take into consideration the routability or the capacity of each channel, or the overall shape of the chip (overloaded channels forced to increase in width could result in a misshapen chip). Also, there is the issue of whether or not the shortest path is natural for the net. For instance, if a bus consists of 16 wires, 8 of them may take one route and 8 others another route, simply because of an obstacle in the way.

3.5. Combined placement and routing

The GAELIC layout package [Loos79a] was designed at the COMPEDA company in England, and it is unique in that it does combined placement and routing. This approach is similar to ours: the placement is not separated from global routing because of the valuable information available to the global router from the placement phase.

The placement problem considered by the GAELIC program is the standard-cell placement problem, with modules of similar size being placed in rows. The program also assumes a predetermined value for one of the final dimensions of the entire layout. This amounts to providing the placement algorithm with a bin to pack, with one of the bin's dimensions specified, namely its width. The algorithm therefore proceeds to place the cells row-by-row in the bin, ending each row when it reaches the wall of the bin. The selection of cells is based on connectivity, starting with bonding pads at the outer edge of the frame and working from one edge, say the bottom edge, inwards. Having placed one row of modules in the bin, global routing is carried out for that row, wires are propagated to the top of the currently placed layer of modules, and then the placement of the next row begins. During the placement, the layout also goes through an improvement phase, where different orientations are tried for each cell.

The global routing is greatly simplified by the assumption of row-wise cell placement. Let's say the algorithm begins with the bottom row. The algorithm proceeds to lay out a row of cells, followed by iterations to choose a good placement that minimizes the total interconnect length in this row. At this stage, all connections to higher rows are ignored (except to provide them with enough channel area to get clear of the current row). The next row of cells floats somewhere above the first row; its exact y-coordinate is not known yet, since the channel routing has not been done for the wires between the rows. Before placement of another row can begin, the interconnections are made, and the channel routing completed immediately above the first row.

As a design approach, this method does not sufficiently satisfy the demands placed on a layout system by current designs. The COMPEDA system is in the class of standard-cell placers and routers, generating a layout very similar to the array topology; this is a more limited problem than the full-custom VLSI placement problem. However, it is a rarity, in that it mirrors our design philosophy: rather than separating placement and global routing into two phases, much can be gained by combining them, although this complicates both problems.

As before, we conclude by comparing our approach with those surveyed. The main purpose of our *incremental* approach to global routing is to supply placement information to the global router. Doing global routing as a separate phase forces the global router to determine a route starting from scratch, whereas with a partial placement, the global router can obtain useful hints about the routing of nets. Moreover, since the partial placement is a smaller subproblem, the effect is to attack the global routing problem in smaller pieces. In addition to obtaining hints for global routing, this approach results in completely specified channels being determined even for a partial placement. Thus, channel routing may also be done incrementally, so that when the placement is completed, there can be no unroutable channels. In other words, the entire algorithm can be guaranteed to complete in time polynomial in the number of modules and nets, with no necessity for either manual intervention or iteration of the placement phase.

Chapter 4

THE GREEDY PLACEMENT ALGORITHM

In the previous chapters, we have seen a range of algorithms for the module placement and global routing problems. While the placement algorithms have varied greatly in their basic principles, they have all had one common characteristic: they are all based on some form of backtracking. For instance, the PI algorithm, reviewed in section 2.1.5, is essentially a straightforward min-cut and global-routing algorithm, in which the backtracking is implemented as a loop all the way back to the placement phase if the channel-routing phase (at the very end of the layout process) detects an undersized or overloaded channel.

The aim of this thesis is to study how well an algorithm can perform in the absence of any form of backtracking. This is the essence of the *greedy* approach: at a given time, any decision that is taken will never be changed in the future. As a result of removing backtracking, we can expect certain tradeoffs. Since the algorithm is not allowed to redo any previously made decision, it will have a bound on its running time, unlike a backtracking algorithm which needs to redo some indeterminable amount of its work during the placement/routing process. On the other hand, since the algorithm cannot go back to change any portion of its placement, it may perform much more poorly than a backtracking algorithm. In the following sections, we develop the basic structure of a greedy algorithm. This basic structure will then be extended in several different directions in order to study the factors that affect the quality of layouts.

We begin with an overview of the block diagram of Fig. 4.1, which gives a top-level view of the stages of the placement process, and the interaction between them. In our algorithm, these stages are not completely independent of each other, as is the case with conventional placement algorithms such as min-cut or hierarchical layout. For instance, we see from Fig. 4.1 that the "Make-WCG and Make-CNG" phase is executed once after each set of clusters has been placed and global routed, whereas in the min-cut approach or in the PI package, the placement phase was completed before any global routing was attempted.

4.1. Preliminary stages

Our placement algorithm begins with the *initialization* stage, by reading data from a CIF description of a set of cells [Mead80a]. We use an extended version of CIF that supports text labels in the layout; these labels are placed on the bounding boxes of the modules to represent terminals that are to be connected to each other. The initialization stage is executed only once for each run of the placer. A detailed set of specifications is available in [Moha87a] Figure 4.2 shows an example layout EX1 that was published as the end-result of a placement-global routing package [Cies87a]. Although this figure has the wire-routing displayed, the input to our placement algorithm would include only a list of modules. In other words, the information obtained from the CIF file would contain only a list of modules, their dimensions, and the positions and numbers of the terminals on each module. Arrows on the nets represent

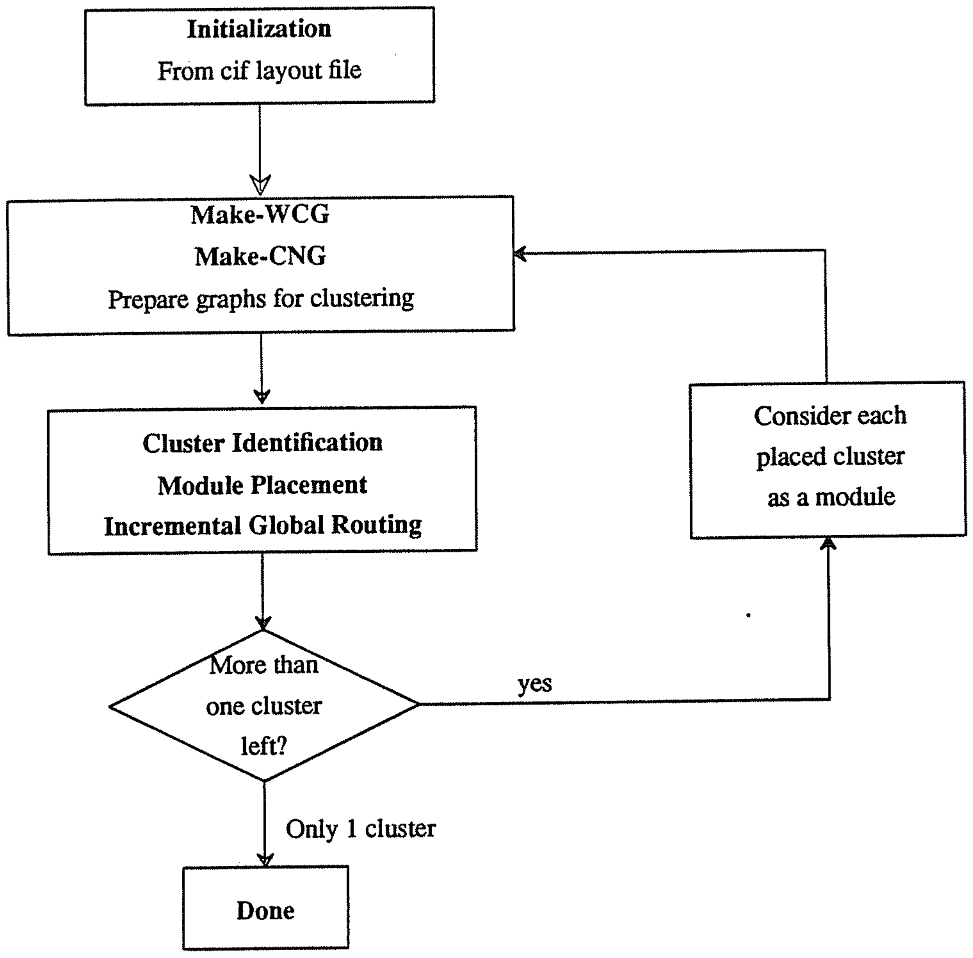

Initialization
From cif layout file
Make-WCG
Make-CNG
Prepare graphs for clustering
Cluster Identification
Module Placement
Incremental Global Routing
More than one cluster left?
yes
Consider each placed cluster as a module
Only 1 cluster
Done

Figure 4.1: Placement algorithm overview

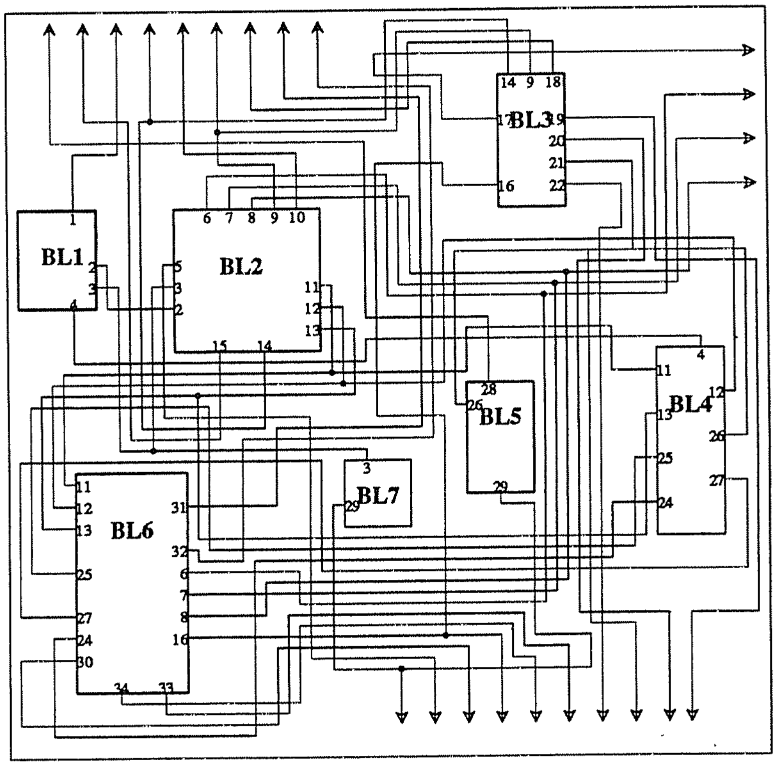


Figure 4.2: Layout example EX1

external nets to I/O pads for off-chip communication.

Following the initialization stage, the algorithm constructs two graphs, the *Weighted Connectivity Graph* (or *WCG*) and the *Closest Neighbor Graph* (or *CNG*). Beginning with the set of modules and their terminals read in during the initialization stage, the WCG is constructed to represent the modules and their interconnections. In

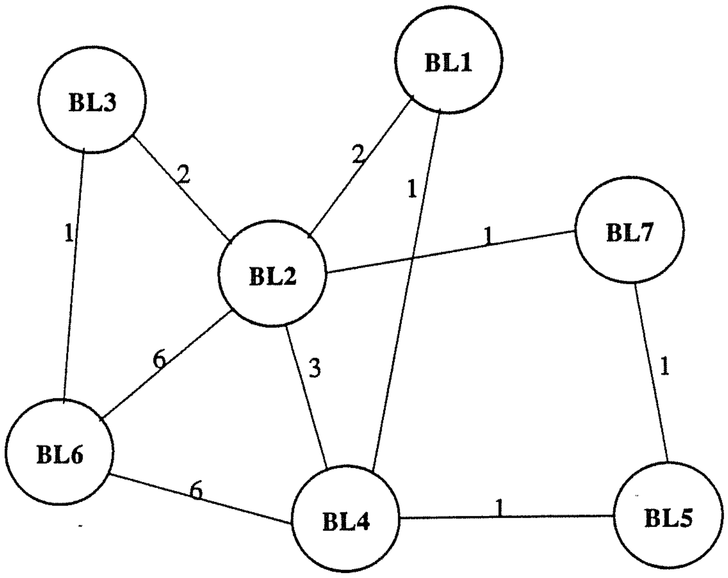


Figure 4.3: Weighted Connectivity Graph for EX1 layout

this graph, modules are represented by vertices, and the connections between them are represented by weighted edges. The weight of the edge between two modules indicates the number of wires between them. Figure 4.3 shows the Weighted Connectivity Graph for layout example EX1. We see that modules BL2 and BL6, for instance, are connected by nets 6, 7, 8, 11, 12 and 13; hence the weight on the WCG edge between BL2 and BL6 is 6. At this stage, the algorithm only considers the number of nets connecting the modules, and ignores the positions of the terminals.

The WCG is the basis on which a Closest Neighbor Graph is constructed. The CNG is a directed graph; it is built by adding directed edges between the vertices of the WCG. An arc from module A to module B indicates that of all the WCG edges incident on A, the one with the greatest weight connects A to B; i.e., A's "most important" neighbor is B. For instance, an arc from module BL4 to module BL6 indicates that, of all the edges of the WCG incident on BL4, the edge of greatest weight

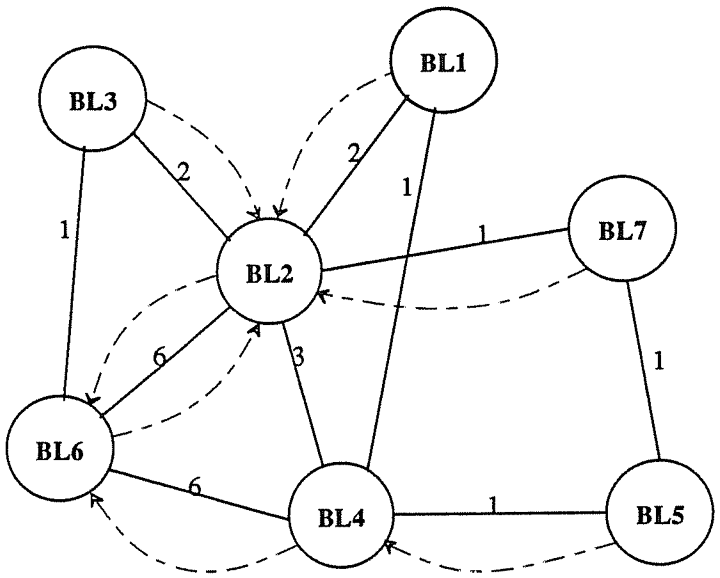


Figure 4.4: Closest Neighbor Graph superimposed on WCG for EX1

connects BL4 to BL6.

4.2. Cluster Identification

In chapter 2, we discussed a few different methods to partition the chip, min-cut and clustering being two of the more popular and important methods. The desirable property of min-cut was that connectivity between members of a cluster was high, and there were few connections between clusters; its drawback was its high computational complexity. Clustering, on the other hand, is simple to implement, and has a low computational complexity, but does not guarantee an even distribution of wires during the global routing phase. In this section, we introduce a novel clustering algorithm which attempts to partition the module-set into clusters, while it also tries to make the placement easy for global routing. We trace the behaviour of our algorithm on layout EX1 of Fig. 4.2.

The clustering stage unfolds as a continuing process of adding modules to a cluster one by one. When the CNG is constructed, it superimposes a directed graph on the WCG; each connected component of the CNG is a cluster. The actual process of selecting modules for placement begins by selecting a *core* module (henceforth referred to as *CORE*) as the most important module in the module set; this effectively selects the most important cluster. The CORE thus selected is the module with the largest in-degree in the CNG. The rationale behind the "largest in-degree" criterion is to pick a module which is the "most important neighbor" to the largest number of modules. From Fig. 4.4, the first CORE selected is BL2, with an in-degree of 4.

We now define a *satellite set*, or *SATSET*, as the set of modules which consider CORE to be their most important neighbor. SATSET is built up from those neighbors of CORE who have WCG edges with weight greater than some threshold, and who have arcs to CORE. In the layout EX1, the SATSET for BL2 would be {BL1, BL3, BL6, BL7}; if the threshold weight for membership in a cluster is at least 1, then BL7 wouldn't be included. The cluster is, therefore, SATSET $\cup$ CORE.

The CNG being one of the major criteria that determine the order in which modules are selected for placement, it is essential that there be no cycle in this graph, since such a cycle could result in an infinite loop during selection of a core module. For instance, suppose that a CNG could be constructed with modules A, B and C having directed arcs between them forming a cycle of length 3. There might then be a cyclic relationship between them: A might be the core module, but at the same time it might be C's satellite. Fortunately, it is easily shown that cycles of length greater than 2 cannot exist in a CNG. Suppose that the CNG contained a set of $k > 2$ vertices M_1, M_2, ..., M_k, and a set of k arcs (M_i, M_{i+1}), $i = 1, 2, ..., k-1$, and (M_k, M_1). Let $W(A, B)$ represent the weight of the edge (A, B) in the WCG. Since M_1 is connected to both M_2 and M_k, the arc (M_1, M_2) indicates that the weight of that WCG edge is greater than the weights of all other WCG edges incident on M_1; specifically, that it is greater than $W(M_1, M_k)$. By a similar argument, it is clear that for all $i = 2, 3, ..., k-1$, the $W(M_i, M_{i+1})$ is greater than the weights of all other WCG edges incident on M_i. This leads to the inequality

$$W(M_1, M_2) < W(M_2, M_3) < W(M_3, M_4) ... < W(M_{k-1}, M_k)$$

Thus, we see that

$$W(M_1, M_2) < W(M_{k-1}, M_k).$$

Therefore, it is not possible for the CNG to contain an arc (M_k, M_1), *since that* *would only be possible if*

$W(M_1, M_k) >$ weights of all other edges incident on M_k

and it was previously established that

$W(M_1, M_2) > W(M_1, M_k)$

This establishes that the CNG cannot contain directed cycles longer than 2 edges. A cycle of length 2 may exist, but that is acceptable to our algorithm.

Once SATSET has been determined, a *satellite* module (or *SAT*) is selected from SATSET. The criteria for this selection are: connectivity to CORE, and how closely the preferred side of SAT matches the preferred side of CORE in dimension. This implies that before SAT can be selected, its orientation needs to be determined. In fact, at this point in the algorithm, all possible modules in SATSET are tried out, in all possible orientations; the module that is selected must have a winning combination of connectivity to CORE and a close match between the dimension of its selected CORE-facing side and the dimension of the SAT-facing side of CORE.

It should be noted that the above algorithm for selecting CORE and SAT modules provides several opportunities for introducing heuristic factors. First, varying the threshold weight for inclusion in the cluster provides a mechanism for selectively blocking off less important modules from membership in a cluster. These less important modules are collected into a set of free modules, termed *loose* modules; we deal with these shortly. Second, the selection of CORE from the set of modules is dependent on the criterion for considering a module as "important". This could be varied between such different aspects as the in-degree of the module in the CNG, area of the module, number of terminals on the module, etc.; or it could be constructed as some composite of all the above criteria. Third, selection of the SAT module from SATSET is dependent on connectivity and closeness of fit of the two facing sides of CORE and SAT. Again, these two criteria could be combined, and the degree of importance of one with respect to the other could be varied to provide a range of results.

A final point to note is that this approach mimics the human designer's approach as outlined in section 2.3. The clustering approach, combined with the threshold connectivity to decide membership in a cluster, is designed to select and place modules approximately in the same sequence as a human designer might choose. The influence of connectivity on the module selection process reflects the human designer's tendency to build the placement influenced by the functional relationship and the flow of control between modules. In switching from placement of modules in clusters to placement of clusters themselves, too, the algorithm draws on the paradigm of the human designer's tendency to build up the placement in clumps of highly connected modules before putting the clumps together.

4.3. Module Placement

The placement stage is, in itself, a complex process. It consists of a combined iterative and recursive process. Iteration is used to repeatedly select satellites and build up the core module; recursion is used whenever a satellite module is too small to appropriately fit beside the core module.

Starting with the Cluster Identification process dealt with in the previous section, the placement process enters the "Recursive placement" phase, as shown in Fig. 4.5. During this phase, all elements of the cluster are placed relative to each

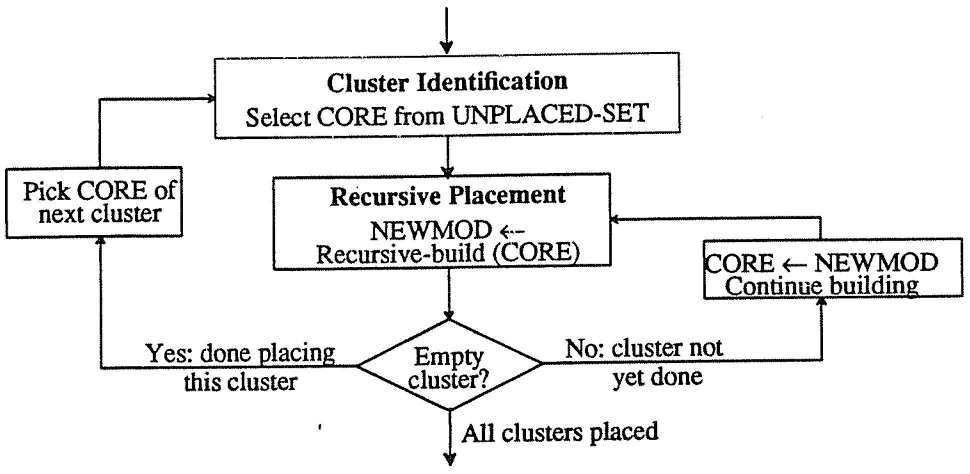


Figure 4.5: Placement phase

other, and routing of wires is done within the cluster. This brings the layout to a state in which terminals to be connected to as yet unplaced modules are available at the boundary of the cluster. The *Recursive-build* algorithm is explained in section 4.4 which follows this section.

A close examination of Fig. 4.5 shows that, during the placement of a cluster, **Recursive-build** is repeatedly called; at each call, the newly constructed module (or *NEWMOD*) replaces the previously selected CORE as the new core module. Laying out a cluster thus turns out to be a process of attaching satellite modules from the cluster, one by one, and then carrying out the necessary wire-routing. After each invocation of *Recursive-build*, control either returns to the recursive placement block (to continue placing modules of the cluster relative to CORE), or if the cluster has been entirely placed, control goes back to the Cluster Identification phase. This latter action causes another CORE to be selected out of the set of remaining unplaced modules (i.e., UNPLACED-SET), and the recursive placement of that CORE and its cluster continues.

Referring back to Fig. 4.1, we see that the placement of all the clusters results in control returning to the preprocessing phase to re-make the two graphs WCG and CNG. In the new WCG, each vertex is a *supermodule* containing an entire cluster, rather than a single module. The effect is thus a staged decrease in connectivity between modules in a cluster. Each invocation of *Make-WCG* would cause a decrease in the degree of connectivity between modules relative to their size, since the highly-connected modules would have been placed together within a supermodule.

To complete the picture, we need to explain the role of *loose* modules. These are the modules that are weakly connected to other modules. Modules whose connectivity lies below a threshold are not allowed to become members of a cluster, and are thrown into a pool of free or loose modules, from which any cluster may draw modules for placement. During the placement of a cluster, if all its elements have already been placed and the Recursive-build algorithm still requires additional modules to place beside CORE, then in that situation a loose module may be selected to fill in the empty space within the supermodule being built. Owing to their low connectivity, these modules typically get oriented and selected primarily on the basis of size rather than connectivity.

4.4. Recursive-build algorithm

The recursive algorithm forms the heart of the strategy for placing modules relative to each other. Figure 4.6 depicts the behaviour of *Recursive-build* with *CORE* as

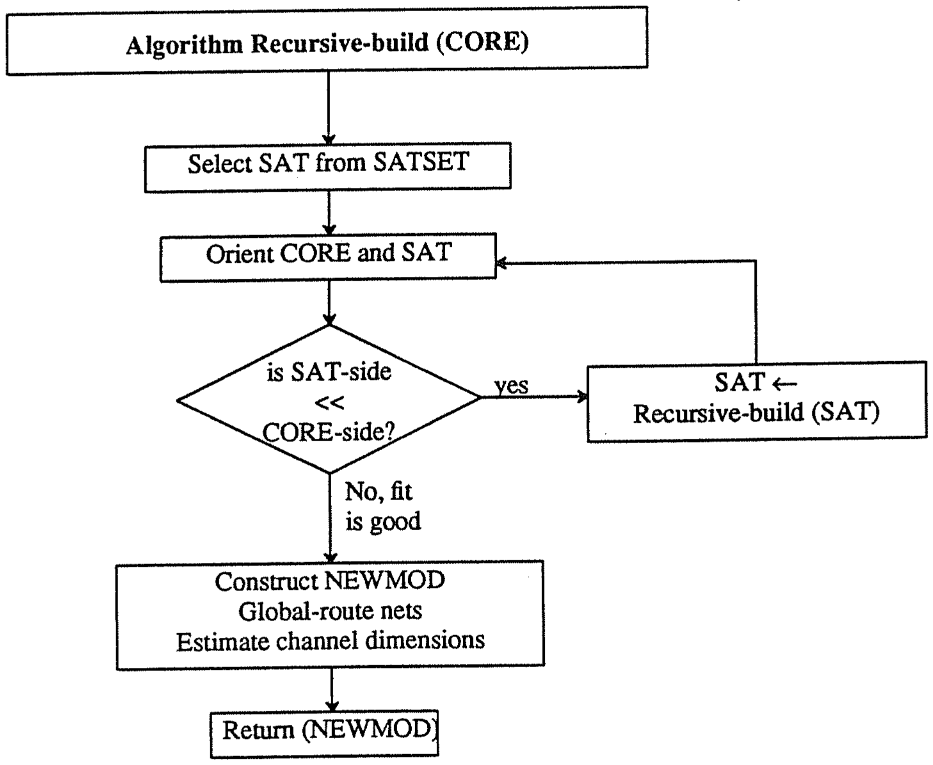

Algorithm Recursive-build (CORE)
Select SAT from SATSET
Orient CORE and SAT
is SAT-side << CORE-side?
yes
SAT ← Recursive-build (SAT)
No, fit is good
Construct NEWMOD
Global-route nets
Estimate channel dimensions
Return (NEWMOD)

Figure 4.6: Recursive module placement

its parameter. At the beginning of the recursive building, a satellite module SAT is selected from SATSET, and a side (SAT-side) is selected for SAT as the preferred side to be oriented facing CORE so as to minimize the width of the channel. Similarly, a preferred side is chosen for CORE, called CORE-side, to face SAT.

The next step is the branch which determines whether this CORE-SAT combination is satisfactory from the point of view of their relative sizes. If SAT-side is smaller than some preset fraction of CORE-side, then Recursive-build is invoked recursively, with SAT as its argument. This causes the process to be executed, but this time SAT is treated as though it were a CORE module. As a result, when the recursive invocation ends, the returned module (consisting of SAT coalesced with other members of the cluster) replaces SAT as the new — and larger — satellite. At this point, control goes to the orienting stage, where the new SAT is oriented.

Since this algorithm is recursive, it is possible that during its invocation with SAT as its argument, it could enter further recursive invocations in order to build up the modules selected to be placed beside SAT. This branching to the recursive call is repeated until all the satellite modules selected during the sequence of recursive calls finally reach a satisfactory size through repeated recursive building-up. Once that point is reached, the non-recursive side of the "is SAT-side < CORE-side?" conditional branch will be taken, leading to the construction of the new module (or NEWMOD), and the routing of wires within it, as is described in the next chapter. The new coalesced module NEWMOD, possibly consisting of many modules, is returned by the algorithm as its return value.

Note that although the CNG is the main criterion for determining the sequence of selection of modules for placement, the iterative process of selecting a module is bypassed by the recursive step. The satellite selection function may be invoked either from the iterative loop, or from within the recursive module-building function. This is the factor that controls the relative positions of the modules, and also the factor that yields densely packed layouts. Note also that this particular style of recursive building of the SAT module is motivated by our desire to construct rectangular modules, and to keep the supermodules from containing a lot of vacant space. In the absence of recursive building, as the CORE module grows larger, placing a small SAT next to it and enclosing the pair within a rectangular boundary causes the area to grow to unacceptable levels. This point is illustrated by the step-by-step placement of layout example EX1 below.

In tracing the placement of example EX1, we refer back to Figures 4.2, 4.3 and 4.4. All positional references to various sides of specific modules (such as the "top of module BL3") are with respect to the original positions of the modules as in Fig. 4.2.

We begin with the selection of a core module. The CNG of Fig. 4.4 shows module BL2 to be the first core module. The satellite set for BL2 includes BL1, BL3, BL6 and BL7; of these, BL6 having the heaviest edge to BL2 (with a weight of 6), it is selected as the first satellite to be placed beside BL2. Module BL6 is deleted from the satellite set, and satellite BL4 of module BL6 is added to it, since BL6 is now in the process of being coalesced with the core module. The orientation is decided: it sets the top side of BL6 to face the right side of BL2 so as to minimize the width of the channel between them. Since the dimensions of this pair of sides do not match very well, recursive building is invoked on the satellite BL6 to try to build it up to size. This restarts Recursive-build with BL6 treated as the CORE parameter, and a satellite is selected for BL6. Again referring to Fig. 4.4, we see that of the remaining satellites, BL4 has the heaviest weight of 4, and hence it becomes the next satellite. BL4 and BL6 are placed and oriented; in this case, the left side of BL6 is selected to face the left side of BL4, having 4 nets in common between these two sides. Since the dimensions match well enough to pass the degree-of-fit threshold, BL6 and BL4 are placed together, the global routing is carried out for the wires in the channel (this process is discussed in detail in Ch. 5) and the two modules replaced by a new module named m8-6-4 (the name is generated from the module numbers of the subcells comprising the new module).

At this point, BL2 and m8-6-4 are to be placed together. Since the dimensions of m8-6-4 are now greater than BL2, they are placed together, forming another new module named m9-2-8. Continuing the satellite selection process, BL3 is the next choice; being too small, it is first recursively built up with BL1 (forming new module m10-3-1), which is then placed next to m9-2-8. That ends the placement of the first cluster. The second cluster contains only modules BL5 and BL7; and finally, the two clusters are placed together to yield the complete layout.

It should now be clear that as the core grows larger and larger, the mismatch between core dimension and satellite dimension also grows, since the core module is being repeatedly built up, whereas satellites are raw modules. This motivates the recursive-build philosophy: as mentioned earlier, if there were no recursive building, the mismatch would cause large areas of the layout to remain unoccupied. The more the number of modules, the larger the core modules would grow, and the greater would be the amount of space wasted. The recursive process is a tool to check that tendency by building up satellite modules to comparable dimensions before coalescing them with core modules.

To sum up the greedy placement approach: we have designed a method for placing modules on a chip without resorting to backtracking. Hence the decisions we make are based on incomplete information. In spite of this constraint on our layout, the algorithm is capable of placing and orienting modules in a way that is helpful to future global routing. The algorithm is also capable of packing modules close together without allowing too much vacant space within the layout. In addition to its performance on the layout, it is a fast algorithm, with a polynomial time complexity and guaranteed completion of the layout process, unlike many industrial packages which may leave some percentage of the work to human hands.

Chapter 5

THE INCREMENTAL GLOBAL ROUTER

The development of the greedy approach to placement leads to certain constraints on the global routing of wires. The philosophy of greed is to make placement decisions with limited information and to avoid changing those decisions at a later stage. Referring back to Chapter 4, we saw that the final layout was built up in stages, with modules being coalesced into supermodules at each stage. In addition, there was no backtracking during the building of the layout. To be consistent with such a philosophy, the global router cannot itself require backtracking, since that would offset the benefits of greed. The constraints on the global routing algorithm are:

(1) Since every call to the ***Recursive-build*** algorithm results in building one pair of modules into a supermodule, the width of the channel between the two modules needs to be determined before recursive building is completed. If this computation were left to be done later, then by the time the channel width is determined, the layout process may have built up many supermodules; it would therefore require changes in the sizes of previously built supermodules in order to insert channels of the appropriate size.

(2) The relative positions of two modules must be decided in order to compute the channel width. The routing of wires in the channel depends on the exact

positions of two modules, in conjunction with other factors such as positions of terminals and detailed routing of wires.

(3) The routing of external wires from within the channel to the supermodule boundary has to be decided. This is a greedy decision: the recursive-build algorithm has no advance information about the future positions of the neighbors of the supermodule currently being built. The global routing algorithm, therefore, needs to route the channel wires "blind", without knowing where the placement algorithm will eventually place the modules to which the nets need to be routed.

(4) The global routing may remain incomplete at the time of module construction. Consider a net N connecting modules A and B. If these modules belong to different clusters, then they will be built into separate supermodules, perhaps many times (because of the recursive-build step), before their supermodules are coalesced. Until that time, as the supermodules containing A and B grow, the net N must be extended to the boundaries of all the intermediate supermodules. This ensures that when the supermodules containing A and B are finally put together, the global routing of net N would already have been partially done; the only routing task left would be to connect the points reached by N on the boundaries of the two supermodules. In other words, nets are global routed *incrementally*, until the placement algorithm puts all the modules connected to a net together in one supermodule. At that point the entire routing area that will be spanned by the nets will lie within the supermodule being built, and the routing can be completed.

Our approach to this problem of greedy global routing is to construct an algorithm which will route wires in directions which will probably be the best direction for the wires, based on the connectivity between the sides of the supermodule and its neighbors. From point number 4 of the above discussion, it is clear that net *N* may not need to be extended to the boundary of the supermodule containing it. If *N* connects only modules within a supermodule, and does not need to be connected to other modules, then it will cease to be a net of interest during the further stages of the layout process. Only if a net is to be routed to other modules external to the current supermodule will it appear as one of its terminals on its boundary.

5.1. Specific goals of the global router

We now define the exact global routing problem to be solved, and the goals of the global router. To begin with, we know that at the end of a module-building step, the relative orientations of CORE and SAT modules have already been determined. The factors influencing this decision are:

(1) The connectivity between pairs of edges, i.e., the size of the intersection between sets of terminals on the facing sides;

(2) The degree of fit— how closely the dimensions of the facing sides of CORE and SAT match;

(3) A penalty for a large percentage of non-matching terminals on the facing sides. This is to encourage the selection of facing sides to reduce the channel width by choosing sides to face each other which will need to route fewer nets in the channel.

Recall that when the global router is called by the module-building algorithm, the sides of CORE and SAT that are to face each other have already been selected. When control returns from the global router, the module-builder proceeds to construct a new module, with complete specification of its internals. According to our definition of a module (in chapter 1), the new module must be a rectangle, with terminals on its boundary. Knowing the starting and final states, the job of the global router can now be defined to be the making of all decisions that will allow CORE and SAT to be placed within a rectangle, and the routing of all nets to be performed within the rectangle bounding the two modules.

We first consider the question of the relative positioning of CORE and SAT modules, and then go on to discuss the routing of nets in the channel and of other nets between the modules.

5.2. Relative positions of CORE and SAT

Initially, when global routing begins, the orientation of CORE and SAT have been determined. However, that only decided which side of CORE and SAT will face each other. The two modules can still be allowed to "slide" relative to each other, with their faces parallel to the channel. The goal in determining a final relative position for the modules is to minimize the width of the channel. However, since the channel width can only be determined by actually doing the global routing [Szym85a], we fall back on greedy heuristics to arrive at a relative position for the two modules.

The heuristic is based on the fact that modules are often designed to be fitted together exactly. The terminals on such modules are usually spaced identically on the sides to be fitted together, thus allowing the modules to be placed next to each other very close together with minimal routing necessary. The terminals on the facing sides are said to have a *matching pitch*, and typically such sets of terminals can be connected by a set of straight-line wire segments, or by a river-routing algorithm (i.e., a routing algorithm which connects pairs of terminals with no crossing wires). Our heuristic is to scan the facing sides of CORE and SAT for a large block of pitch-matched terminals, and to pick the relative position which maximizes the block of pitch-matched terminals. This is determined by repeatedly sliding CORE and SAT with respect to each other, and counting the number of matched terminals at each position. In the absence of matching pitches, the modules are positioned with their centers aligned.

5.3. Channel net routing

Consider a set of nets in the channel, i.e., on the facing sides of CORE and SAT. The direction of routing of these nets to the boundary of the CORE-SAT supermodule will later contribute to an orientation decision when the supermodule is placed beside one of its neighbors. Hence, the goal in global routing the channel nets is to cause this routing to fit in harmoniously with the existing terminal positions on the other sides of CORE and SAT.

We illustrate our approach with the example of Fig. 5.1. Modules A and B are to be coalesced into new module AB. We term the set of nets on the facing sides as *CHANNETS*, after their existence in the channel. In our example, Channets = {2, 4, 5, 6, 7, 8, 9, 11, 13}. The set of nets on the top-side of AB (or *TOPnets*) is {2, 10, 13, 16, 19, 25}; and similarly, *BOTnets* is {4, 8, 9, 23, 27}. The sets of nets to be routed

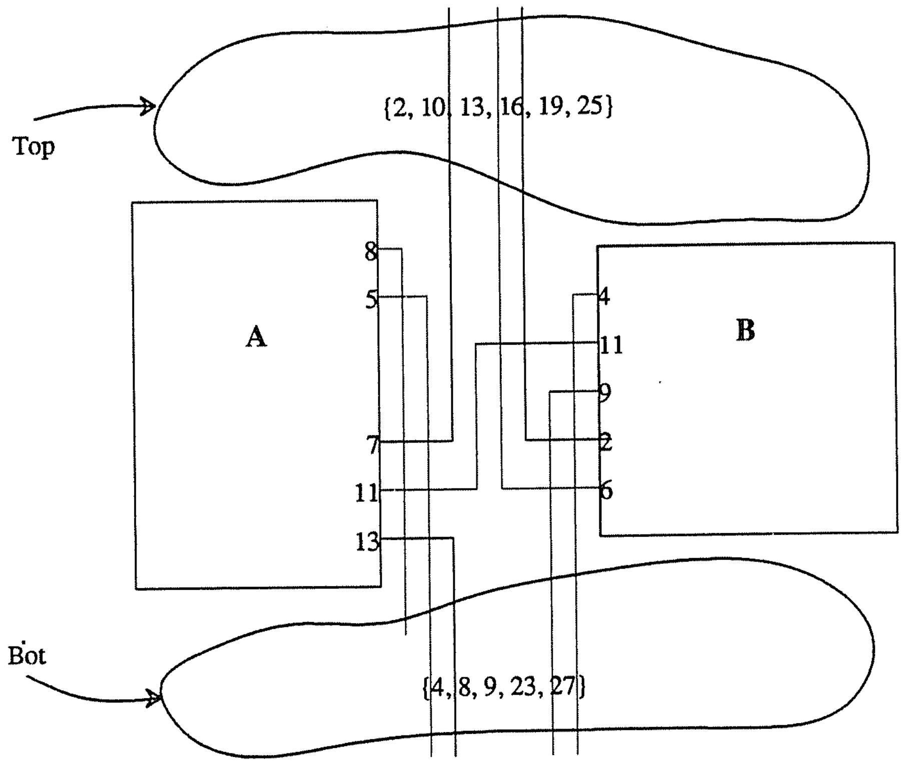


Figure 5.1: Global routing of channel nets.

to various neighbors is also determined to be the following sets:

Channel nets to neighbor 1 = NBRnet{$_1$ = {1, 4, 9};

NBRnets$_2$ = {1, 5, 9, 13}

NBRnets$_3$ = {2, 14, 27}

$\text{NBRnets}_4 = \{6, 7, 19, 25\}$

$\text{NBRnets}_5 = \{10, 27\}$

The set of nets in the channel contains some nets that only need to be to be routed between the modules{A and B; plus some ***external nets*** which are to be routed to the boundary of the supermodule AB. The key decision at this stage is which side — top or bottom — to choose for global routing each external net in Channets. To motivate our algorithm, it should be noted that the positioning of a neighbor X will be decided by the nets on the other sides of A and B, in addition to other factors. If neighbor X has a large intersection with BOTnets, it would make sense to route all channets connected to X towards the bottom of AB. If some of those nets were also connected to neighbor Y, then it would be reasonable to assume that, were Y to be placed first (before X) then the Channets connected to Y would also have to be routed towards the bottom of AB. In other words, we need to partition the set Channets into equivalence classes.

To begin with, let AB have k neighbors NBR_1 ... NBR_k. For each neighbor NBR_i, we have a set of nets C_i going from Channets to NBR_i, i.e.,

$C_i = NBRnets_i \cap \text{Channets}$

These subsets of channel nets C_1 .. C_k are the starting point for partitioning them into equivalence classes according to the equivalence relation:

$C_i \equiv C_j \textbf{ iff } C_i \cap C_j \neq \emptyset$

In other words, if two neighbors NBR_x and NBR_y are both connected to net N, i.e., if N is to be found in subsets C_x and C_y of Channets, then both C_x and C_y should get routed in the same direction. Hence C_x and C_y should belong to the same partition or equivalence class.

The equivalence class partitions can then be decided separately. Given partition P, connected to, say, a group of neighbors G, the direction of routing of P is determined by comparing the sizes of intersection of TOPnets and BOTnets (i.e., nets on the top and bottom sides of supermodule AB) with the netlists of the neighbors in the group G. If neighbor group G has a greater intersection with TOPnets then all Channets in partition P should be routed towards the top, and *vice versa*.

To summarize, the goal of the global routing algorithm is to keep down the channel width and the total wiring length. Channel width is determined based on maximizing the set of pitch-matched terminals on the facing sides of the modules. Wiring length is limited by making a greedy routing decision that tends to keep nets to neighbors from being routed all the way around the module. The heuristic used to pursue this goal is the routing of subsets of nets in the channel in the direction that the neighbors connected to the nets are likely to end up on.

5.4. Routing non-channel nets

The only nets yet to be routed are those which connect CORE and SAT modules. These nets are routed in a straightforward manner by computing intersections of nets on pairs of sides of CORE and SAT modules, and setting aside enough tracks to accommodate the number of nets expected. For instance, in routing a net from the left side of module A to the top side of module B, the intersection of the left-nets of A with TOPnets would yield the number of nets traveling between those two sides. It would then suffice to compute the amount of routing area to set aside for routing that number of nets. This process is repeated for various pairs of sides; nets that may choose either direction are split into halves, and each half is routed in different directions to reduce the routing area width.

5.5. Detailed channel routing

As we had pointed out in Chapter 1, we do not include detailed channel routing in our algorithm. Instead, using the information collected and computed in the above sections, the channel width is estimated. This computation is based on the maximum channel density (see Chapter 1), and on the maximum block of terminals on both of the facing edges whose positions match exactly. Recall that the density at any point in a channel is defined as the number of nets crossing that point. The maximum channel density can therefore be considered to be an upper limit on the width of the channel, since at some point in the channel, there are enough nets needing that width of channel.

The channel width estimation begins by determining, from the previous global routing steps, the sets of nets entering the channel from either end. A straight scan of the channel is sufficient to calculate the maximum channel density, by keeping track of the number of nets on an *open* list. Membership of a net N in this list implies that N has not yet been completely routed in the channel. For instance, if N were positioned at column 4 on the top edge of the channel, and at column 17 on the bottom edge, then it would enter the *open* list when column 4 is considered, and leave the list at column 17. Nets entering at the left edge of the channel are entered into the *open* list initially; nets leaving the channel at the right edge remain in the *open* list after the scan is complete.

As we have pointed out in Chapter 1, there are many excellent algorithms for detailed channel routing. Many of these algorithms have also exhibited the ability of routing channels using only one track more than the channel density. We have therefore chosen to estimate the channel density, and allow an additional 15% space, and have concentrated rather on the algorithms for placement and global routing[1]. It should be noted that the wiring space provided is a liberal estimate of the spacing required. It is quite likely that an additional compaction stage may produce significant area gains if applied after completion of the layout. We deal with this issue in Chapter 7.

The figure of 15% was arrived at after noticing that a majority of the channels contained fewer than about 10 nets to be routed. The figure for the additional channel space percent had to be high enough to provide at least one additional track; less than 15% allowance causes too small an increase in channel width to accommodate even one additional track.

Chapter 6

RESULTS

The algorithms described in the previous two chapters comprise the greedy placer and incremental global router; we henceforth refer to the package as the Placer. The Placer has been implemented in Lisp, and produces layouts with positions of modules and routing areas specified. A Placer layout run producing a single layout took between 36 seconds for the smallest layout, to 166 seconds for the largest; timing was on a VAX 11/750 running 4.3 BSD Unix[2] including the time for operating system overheads. Input to the Placer is in the form of a *CIF* file [Mead80a], in which each module is represented by a *cell* consisting of a rectangle of metal with numeric labels on the boundary to denote terminals. The Placer produces a CIF file as output, containing a subcell for each application of the Recursive-build function. These subcells are organized in levels, with one level of cells per level of recursion during the Recursive-build process. Subcells contain both rectangles (the constituent modules) and routing areas (denoted for viewing purposes by polysilicon, or red, rectangles). The Placer also outputs the area, estimated total wiring length, and the aspect ratio of

UNIX is a trade-mark of Bell Telephone Laboratories the finished layout to a separate output file.

In the following sections, we proceed in stages, evaluating the performance of the algorithm with respect to one decision criterion before turning to the next one. The sequence in which we consider the criteria reflects their relative importance to the algorithm. We first review those criteria which we consider to be central and vital to our approach, and which we believe is the contribution of this thesis. These criteria are:

- the size and membership of a cluster; and
- the stage at which recursive building takes place.

The next pair of criteria that we discuss determine the sequence in which modules will be selected for placement, and their orientation:

- the relative importance of connectivity and dimensional fit in determining the next module to be placed;
- the relative importance of connectivity, dimensional fit and non-channel net-count in determining the orientations of a CORE-SAT placement decision.

Finally, we consider the factors that control how the cluster-membership, recursion-threshold and connectivity-weight vary with increasing size of cluster; i.e., as the cluster-building process makes larger and larger supermodules, should the cluster-membership criterion change? Should the recursion threshold change? What should they change to? For instance, a tactic which was suggested in Chapter 4 was to begin with a low value for the threshold dimensional fit that triggered recursion; and to increase it later, as modules became larger. This set of factors is in the nature of "fine-tuning" the Placer.

The test layouts on which we ran the Placer were presented as results in various articles published in the IEEE Transactions on CAD of Integrated Circuits and Systems. Our fist two example layouts (labeled *EX1* and *EX2* in our results) were obtained from [Cies87a]. The placement was carried out by *digraph relaxation*, which begins with a rough initial relative placement of modules. A placement improvement process based on relaxation (discussed in section 2.2) was carried out, combined with a branch-and-bound approach for area minimization. The placement was followed by channel definition and routing stages.

From [Marg87a], we obtained layout *EX3*. This layout was published as an example of a global router that was incorporated into a chip floor-planning tool, ARIANNA [Anto85a].

Finally, we obtained *EX4*, *EX5* and *EX6* from [Roth83a]. These layouts were again published as an example of a global routing algorithm, starting with a given initial placement. In addition to the placement and routing, a compaction stage was also included in their production. We should point out that comparisons between the Placer's layouts and examples *EX3*, *EX4*, *EX5* and *EX6* are therefore rather biased against the Placer, since the Placer has no compaction stage built into it, nor does it have the human interaction and guidance that a floor-planning system (such as the ARIANNA system) has access to.

6.1. Notation

Before presenting the results, a short description of the notation used in the graphs and tables is necessary. The results were obtained by repeatedly running the Placer at different settings of the weight-values for the various decision criteria. As the following sections further show, many of these weights are quite unstructured in terms of their range of values and their relationship to the actual areas obtained. In order to meaningfully compare the performance of the Placer, the runs were repeated with identical sequences of weights on different layout examples. The results are therefore presented with simply the layout run number on the x-axis, rather than the actual weight value. The correspondence between runs and actual weights is tabulated separately for each set of tests.

Against the series of layout numbers, we plot the resulting area, total estimated wire-length, and final aspect ratio. Area (and wire-length) plots are of the ratio of obtained layout area (and wire-length) over the area (wire-length) of the example in the original publication. In other words, the area and wire-length plots all indicate how well the Placer did relative to the original layouts; the $y = 1.0$ line marks the area of the original layout. In the following sections, we see that many layouts produced by the Placer have areas and wire-lengths that fall below this line, signifying that at those points, the Placer has built a layout with smaller area or wire-length than the original (the exact result value indicates what fraction the Placer-generated area or wire-length is relative to the original layout). The aspect ratios of all the original examples were very close to 1; hence we have plotted the absolute aspect-ratio values without modification.

Each set of test results is presented in three parts. First, a table shows the weight values of interest corresponding to each layout number. This is followed by six graphs, one per layout example, showing the behavior of area, wire-length and aspect-ratio against layout number. Area plots are marked by circle symbols (○), wire-length by triangles (Δ) and aspect-ratios by crosses (+). Finally, the ranges of weights corresponding to the best layouts are summarized in a separate table.

6.2. Impact of cluster size on layout

We begin by presenting the effect of varying the cluster size. Recall from section 4.2 that the selection of modules for placement is based on the arcs in the Closest Neighbor Graph (or *CNG*). In fact, during the construction of the CNG (which was dealt with in section 4.1), an arc is inserted only if the weight of the WCG edge exceeds a certain minimum value, which we call the *Cluster-threshold*. The threshold connectivity is specified as a percentage of the average number of terminals per module, which can then be used on any layout independent of the actual number of terminals that the particular layouts may have.

Consider the simple mechanism of providing a threshold connectivity level that will block off weakly-connected satellites from inclusion in a cluster. By varying the cluster-membership threshold, it is possible to test not only the threshold that leads to the best layouts, but also to choose between different *algorithms* themselves. Setting a very low cluster-threshold (approaching zero) leads to the inclusion of *all* possible satellites in a cluster, maximizing the size of the cluster. On the other hand, by setting a very high cluster-threshold (approaching infinity), we have a mechanism to allow *no* members at all in the cluster. In that circumstance, the CORE module selection algorithm would find that all modules had zero in-degree in the CNG, and would therefore fall back on the conventional measures of area and size of net-list to select the next module for placement.

The table of weights vs. layout numbers is shown in Fig. 6.1. Figures 6.2 — 6.7 show the behavior of the six test layouts at different values of the cluster-threshold. (Besides the cluster-threshold, all other factors were set at approximate values; their best values are established in later tests.) An extremely encouraging sign obtained from this first series of tests is the fact that 4 of the 6 examples were placed and routed in *less than the original areas and wire-lengths*. Although a greedy approach could conceivably produce poorer layouts compared to global approaches, we find that such is not the case; and in the following sections, we see that the remaining two layouts come down in area as wider ranges of weights are tested.

Looking at Fig. 6.2 (the result for layout *EX1*), we see that the best layout in terms of both area and wire-length occur at runs 16 through 30. These layouts correspond to cluster-threshold values ranging from 1.5 through 100, i.e., they correspond to layouts obtained when the cluster-threshold value is set to a maximum. In fact, the same layout is obtained for this entire range of cluster-threshold. This indicates that as low a cluster-threshold as 1.5 suffices to block off *all* arcs from being formed. Since this value of cluster-threshold produces the best layout, we must therefore conclude that for this example, using in-degree in the CNG to identify core modules produces layouts with greater areas than those produced by using conventional measures such as module area or module net-list size.

The other examples, however, tell a different story. Example *EX2* (Fig. 6.3) has little variation in area and wire-length, but a very high aspect-ratio on many of the layouts leaves the layouts numbered 8 — 10 and 18 as the best ones. These correspond to cluster-threshold values of 0.7 — 0.9 and 2.0 as the most appropriate values.

Examples *EX3* and *EX5* have their most promising layouts at the minimum cluster-threshold values, i.e., with the largest clusters possible. *EX4* does well at cluster-threshold = 2.5 — 3.0, while *EX6* does well almost everywhere, with layouts 17 — 18 (i.e., cluster-threshold = 1.7 — 2.0) doing marginally better than the others. The results are summarized in Fig. 6.8.

We can thus claim that for a majority of the test layouts (examples *EX2* through *EX6*), the presence of arcs in the CNG — and hence, the use of in-degree as the primary criterion for Core-module selection — yields better layouts. Even in the case of *EX1*, we find that the second-best layouts (numbered 5 — 15, i.e., cluster-threshold = 0.4 — 1.4) are not significantly worse layouts than the best layouts. We therefore conclude that the in-degree criterion — which is one of the key ingredients in our attempt to simulate the human designers' approach — does indeed lead to comparable or better layouts. It should also be noted that these sets of tests have all been made at some plausible pre-determined values for all weight factors. In the following sections, we shall see that as wider ranges of values are tested, the global view results in a changed set of results. However, even as limited a test as this first one has sufficed to bring us to the conclusion that using the in-degree criterion to choose core modules produces better results than the conventional area/net-list criteria.

Layout number	Weight	Layout number	Weight
1	0.01	16	1.5
2	0.1	17	1.7
3	0.2	18	2.0
4	0.3	19	2.5
5	0.4	10	3.0
6	0.5	21	3.5
7	0.6	22	4
8	0.7	23	5
9	0.8	24	7
10	0.9	25	10
11	1.0	26	15
12	1.1	27	20
13	1.2	28	30
14	1.3	29	50
15	1.4	30	100

Figure 6.1: Cluster-threshold weights vs. layout numbers

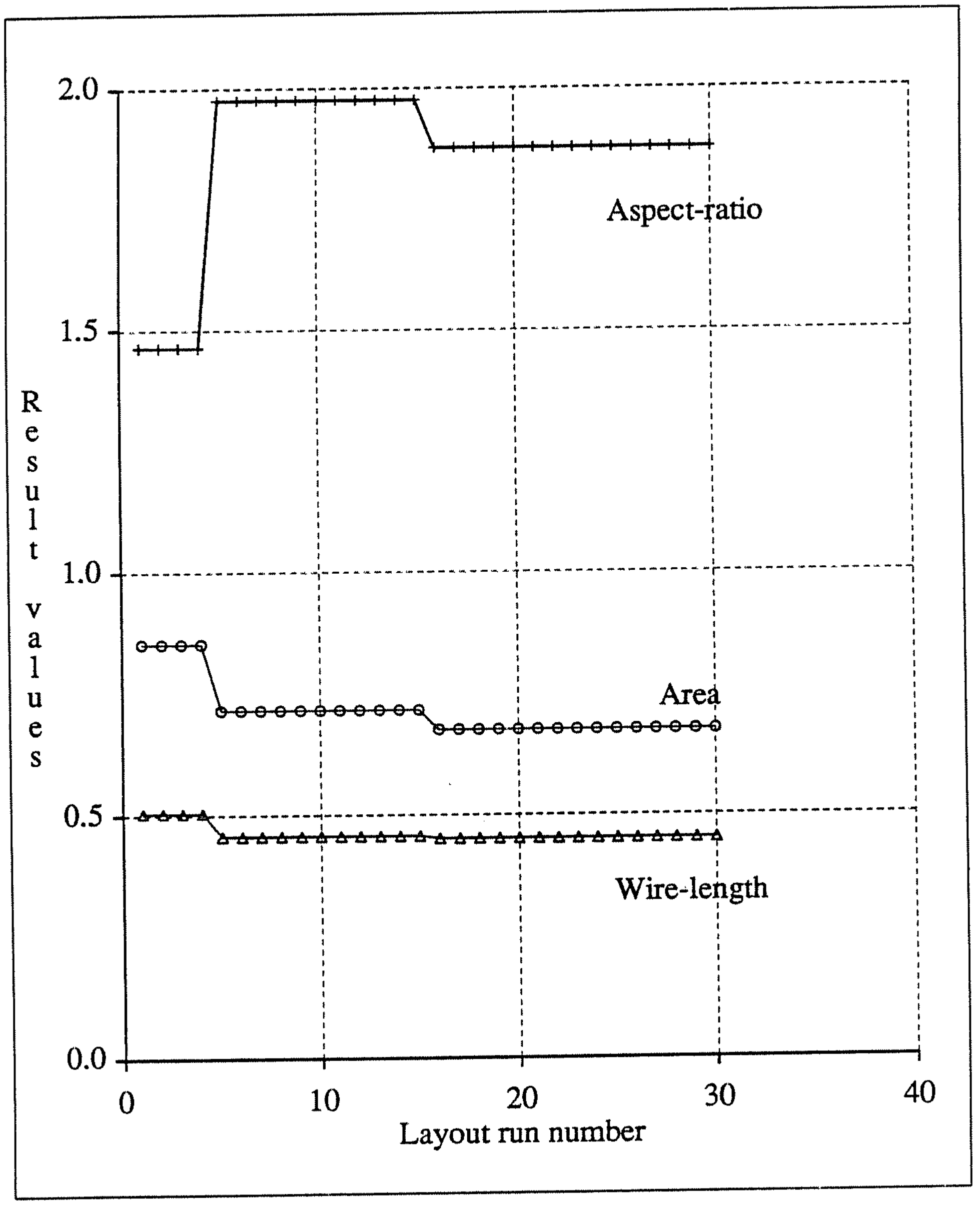


Figure 6.2: Cluster-threshold performance for *EX1*

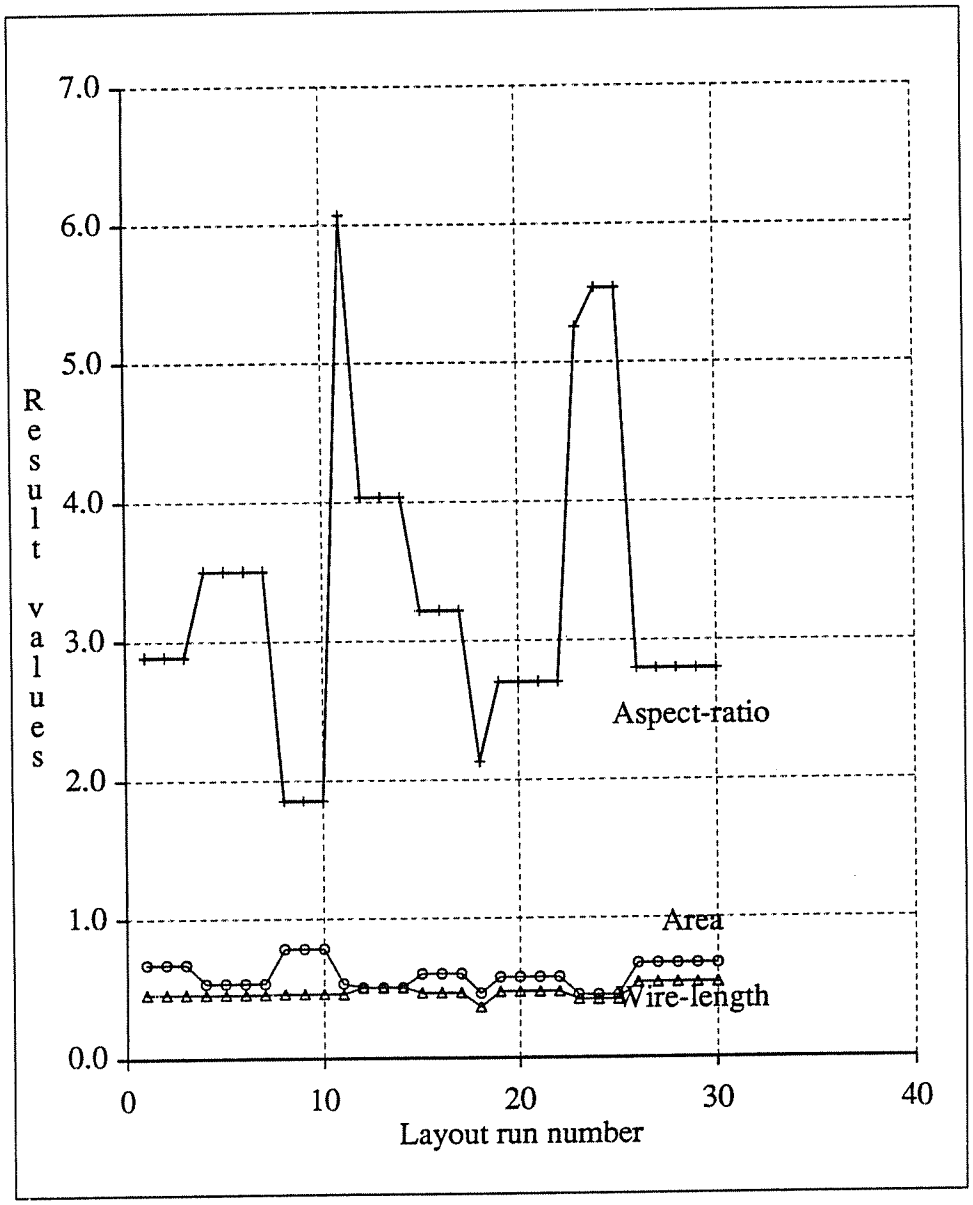


Figure 6.3: Cluster-threshold performance for *EX2*

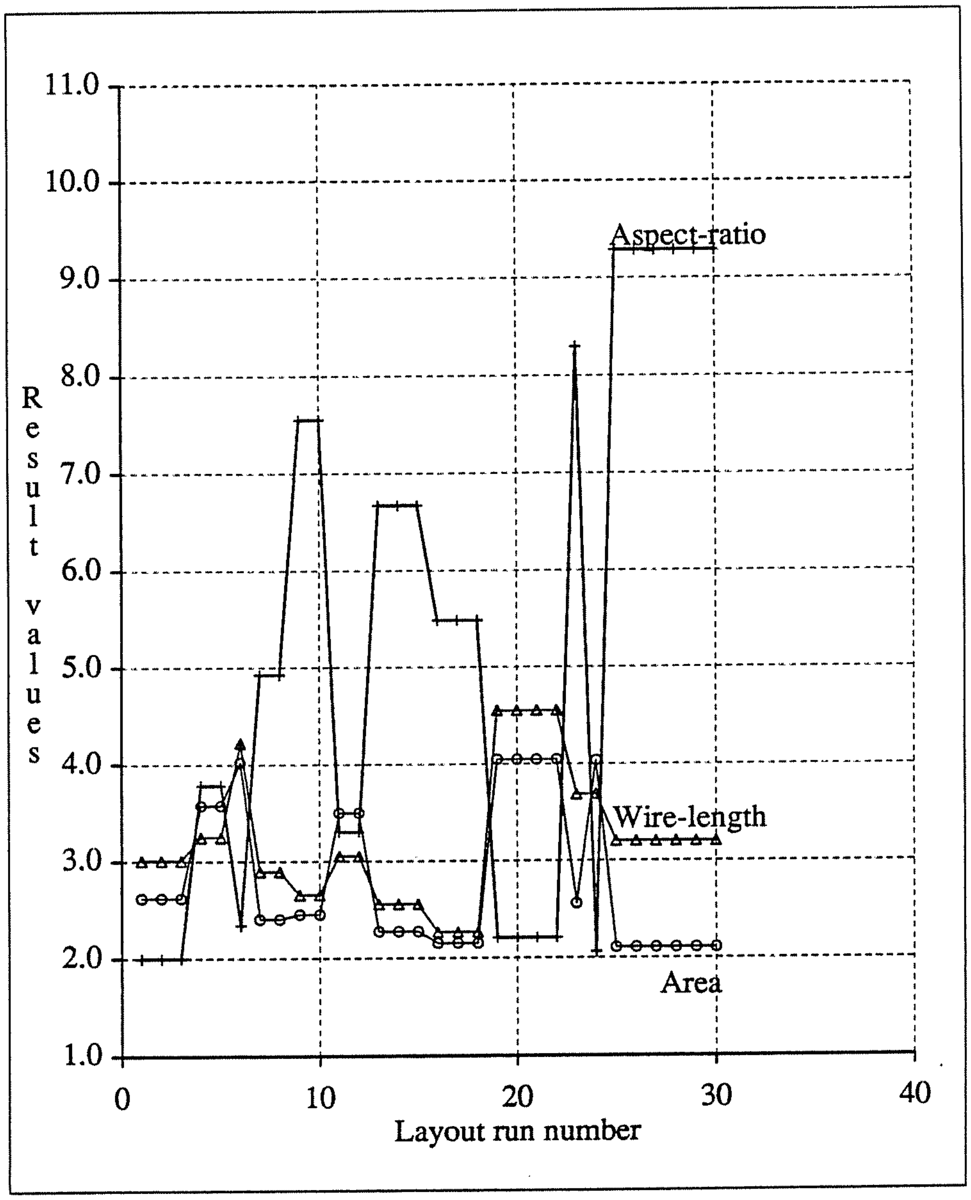


Figure 6.4: Cluster-threshold performance for *EX3*

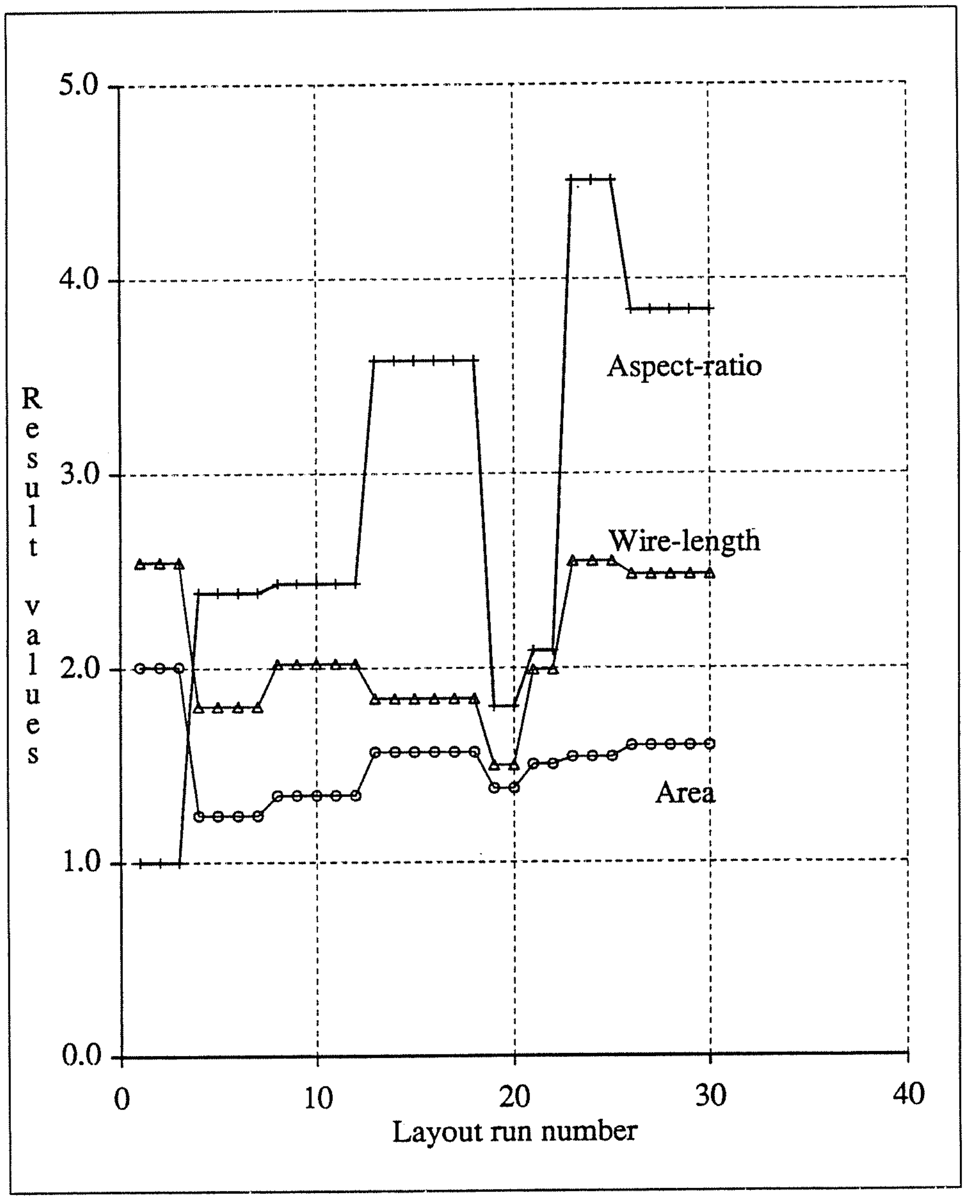


Figure 6.5: Cluster-threshold performance for *EX4*

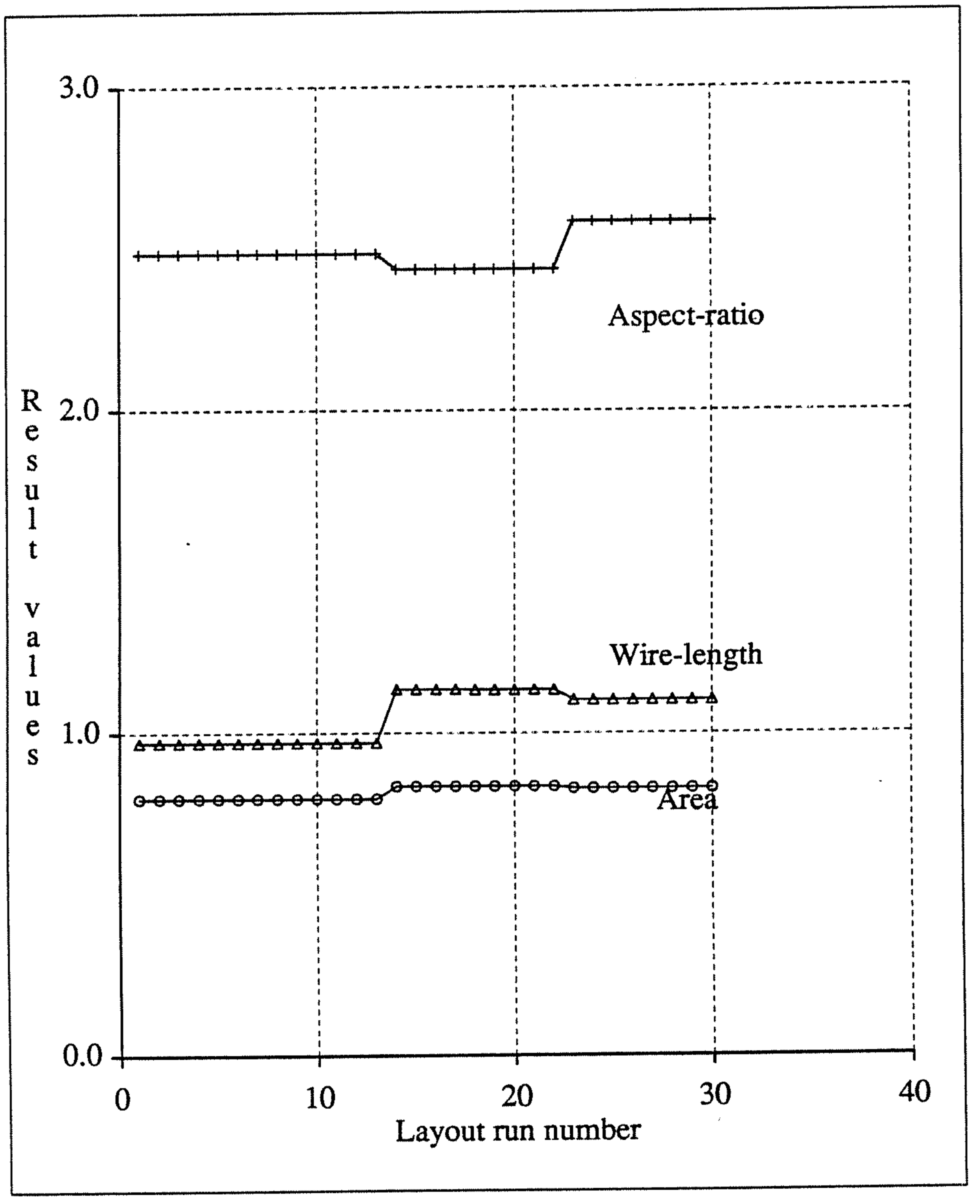


Figure 6.6: Cluster-threshold performance for *EX5*

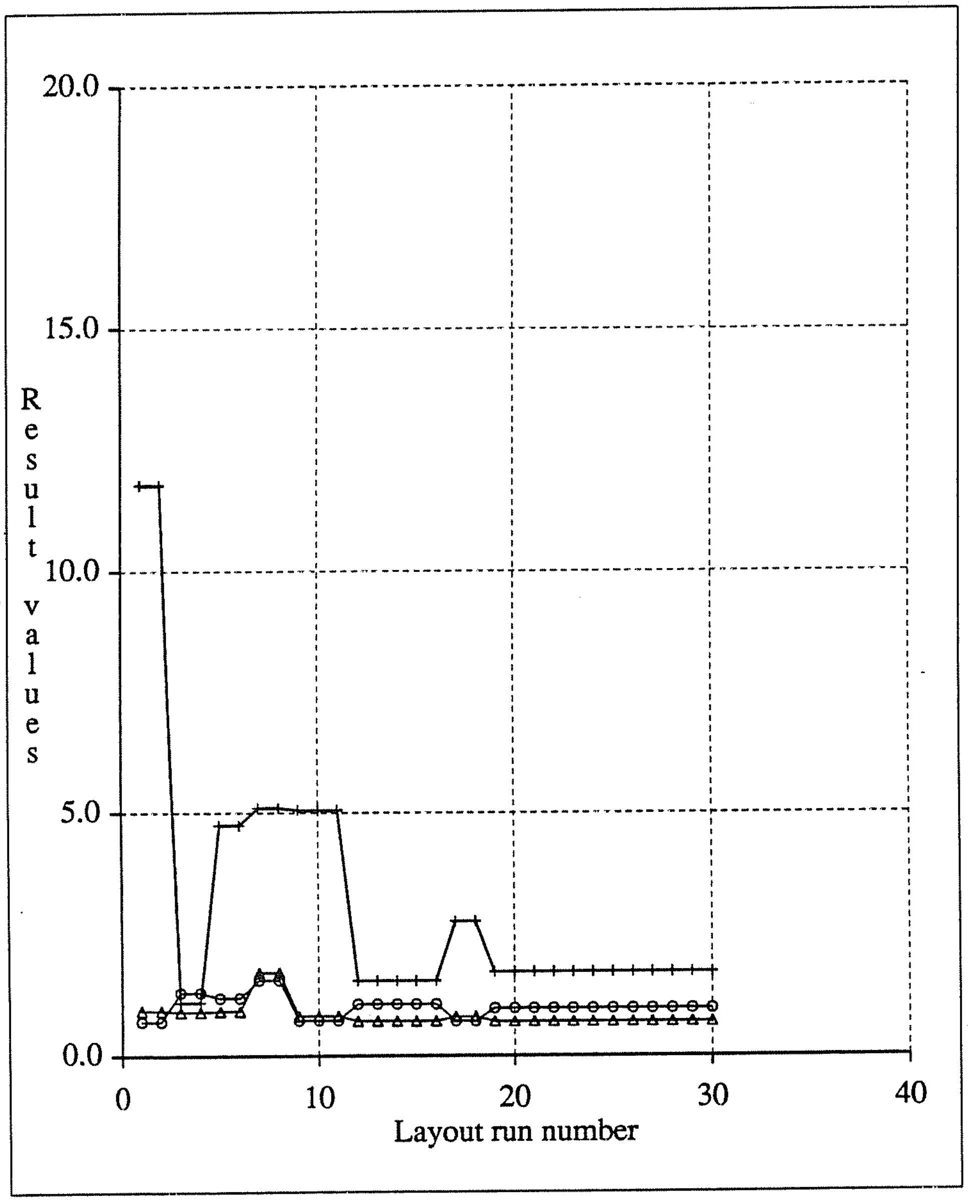


Figure 6.7: Cluster-threshold performance for *EX6*

Layout name	Layout numbers	Cluster-threshold range
EX1	16 — 30	1.5 — ∞
EX2	8 — 10	0.7 — 0.9
EX3	1 — 3	0.01 — 0.2
EX4	19, 20	2.5 — 3.0
EX5	1 — 13	0.01 — 1.2
EX6	17, 18	1.7 — 2.0

Figure 6.8: Summary of the best Cluster-threshold values

6.3. Recursion-threshold: when to stop recursive-building

The first set of tests dealt with one of the central themes of this thesis, the idea of clustering based on the in-degree of a module in the Closest Neighbor Graph. The next factor we consider tests the other key idea contributed by this thesis, namely, the idea of recursively building up satellite modules in a greedy incremental manner. Of course, the entire concept of greedy placement and incremental global routing can only be evaluated ***in toto***, after the best weights for all the decision criteria have been determined. To some extent, we have already seen that greedy placement can do comparably well or better than a global scheme: of the six examples, four have been placed and routed with a lower area, and three of them with a lower wire-length as well.

In this section, however, our testing is aimed at determining a ***recursion-threshold***. The Recursive-build algorithm quits its recursion when the built-up SAT edge dimension falls within a predetermined factor of the CORE edge dimension; this is what we term the recursion-threshold. Setting a low value for the recursion-threshold causes the Placer to quit the recursive-build process early, when SAT is still relatively small compared to CORE; a high value (approaching 1) causes it to build up a SAT recursively until its dimension is very close to or greater than the CORE edge.

Result values have been plotted for a series of recursion-thresholds, which has itself has been iterated over a set of representative cluster-threshold weights obtained from the previous section. This provides a more global view of the behavior of the Placer. Figure 6.9 lists the layout numbers and their corresponding dimension ratios, while Figures 6.10 — 6.15 contain the graphs of result values against layout numbers, and Fig. 6.16 summarizes the conclusions.

Example *EX1* now displays a behavior that is a major shift from the curve of Fig. 6.2 in the previous section. It is now seen that the best layouts for *EX1* are at a cluster-threshold value of 0.8, and not at ∞ as previously concluded; the culprit in the previous conclusion being an assumed value for the recursion-threshold that did not suit *EX1*. The recursion-thresholds that produce the best *EX1* layouts range from 0.01 through 0.5. In other words, the best performance for this example occurs when recursive building carries on with no limits at all!

Similarly, some of the other examples also produce layouts that are better than those obtained in the previous section. A study of the summary (Fig. 6.16) shows the emergence of two broad groups of layouts by the recursion threshold criterion. *EX1* and *EX4* do well with relatively little recursive-building, while the other layouts have their best layouts at high recursion-thresholds. In addition to this, it should also be noted that the role of clustering is now established firmly, with all six examples producing their best layouts at cluster-threshold values ranging from 0.8 to 2.0. The performance at cluster-threshold = ∞ is no longer the best; in other words, clustering using the in-degree criterion from the CNG can be claimed to be a success. Finally, in terms of absolute performance, we have examples *EX1*, *EX2*, *EX5* and *EX6* below the 1.0 mark, and example *EX4* at about the 1.3 mark, which means that for most of the layouts, the Placer is doing much better than the global approaches that produced the original placements.

One other characteristic worth commenting on is the extreme fluctuations in aspect-ratio. This odd behavior is explained by the fact that control over aspect-ratio has been left to an indirect mechanism, namely a combination of connectivity and dimensions in the orientation stage. As a result, in some circumstances, the layout is built as a row of modules, all side-by-side, with very little building in the orthogonal direction, yielding thin and long layouts with very high aspect ratios. We shall have more to say on this subject in section 6.8.

Layout number	Weight	Cluster-threshold
1	0.01	0.01
2	0.1	
3	0.2	
4	0.3	
5	0.35	
6	0.4	
7	0.45	
8	0.5	
9	0.55	
10	0.6	
11	0.65	
12	0.7	
13	0.75	
14	0.8	
15	0.85	
16	0.9	
17	0.93	
18	0.96	
19	0.99	
20 — 38	0.01 — 0.99	0.8
39 — 57	0.01 — 0.99	2.0
58 — 76	0.01 — 0.99	2.5
77 — 95	0.01 — 0.99	∞

Figure 6.9: Recursion-thresholds vs. layout numbers

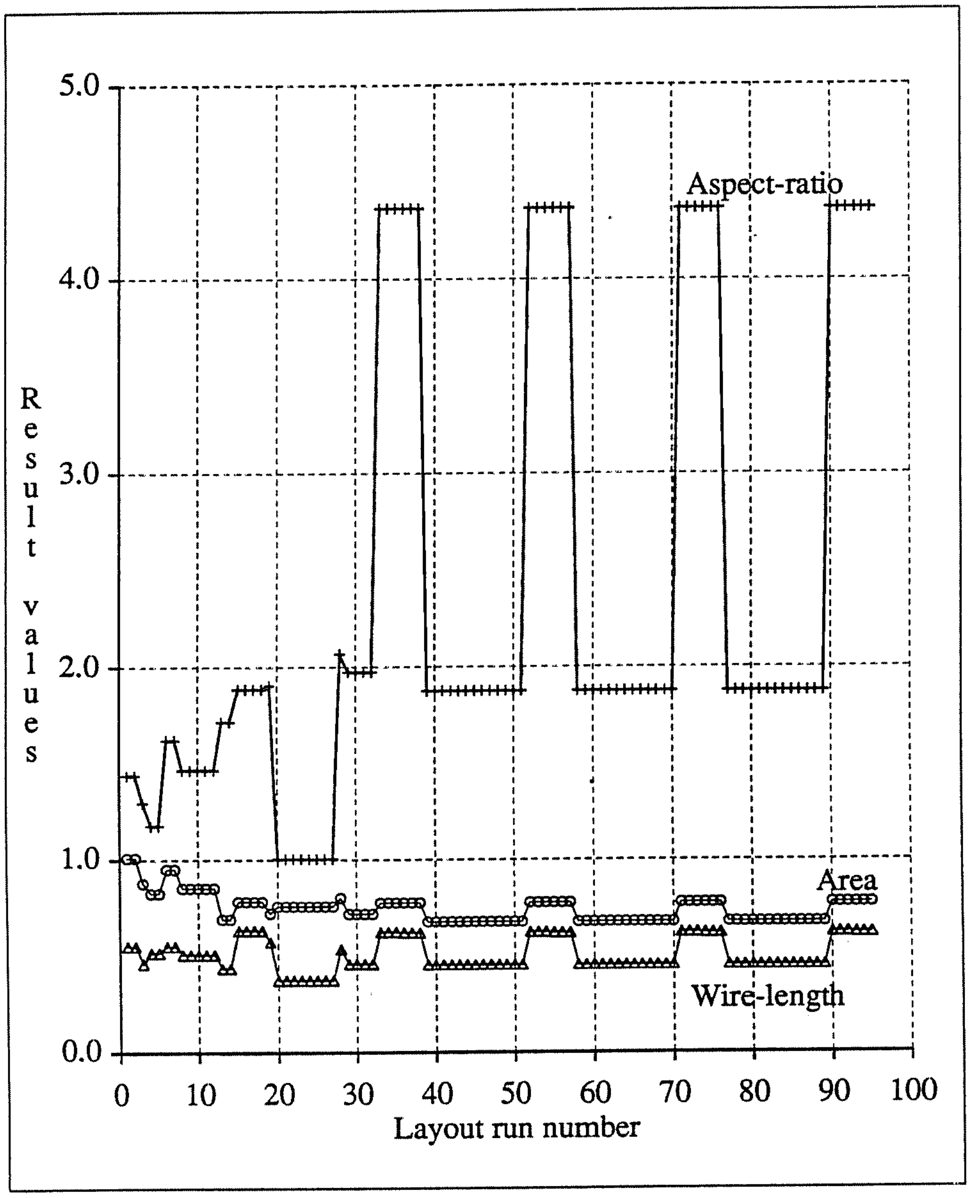


Figure 6.10: Recursion-threshold performance for *EX1*

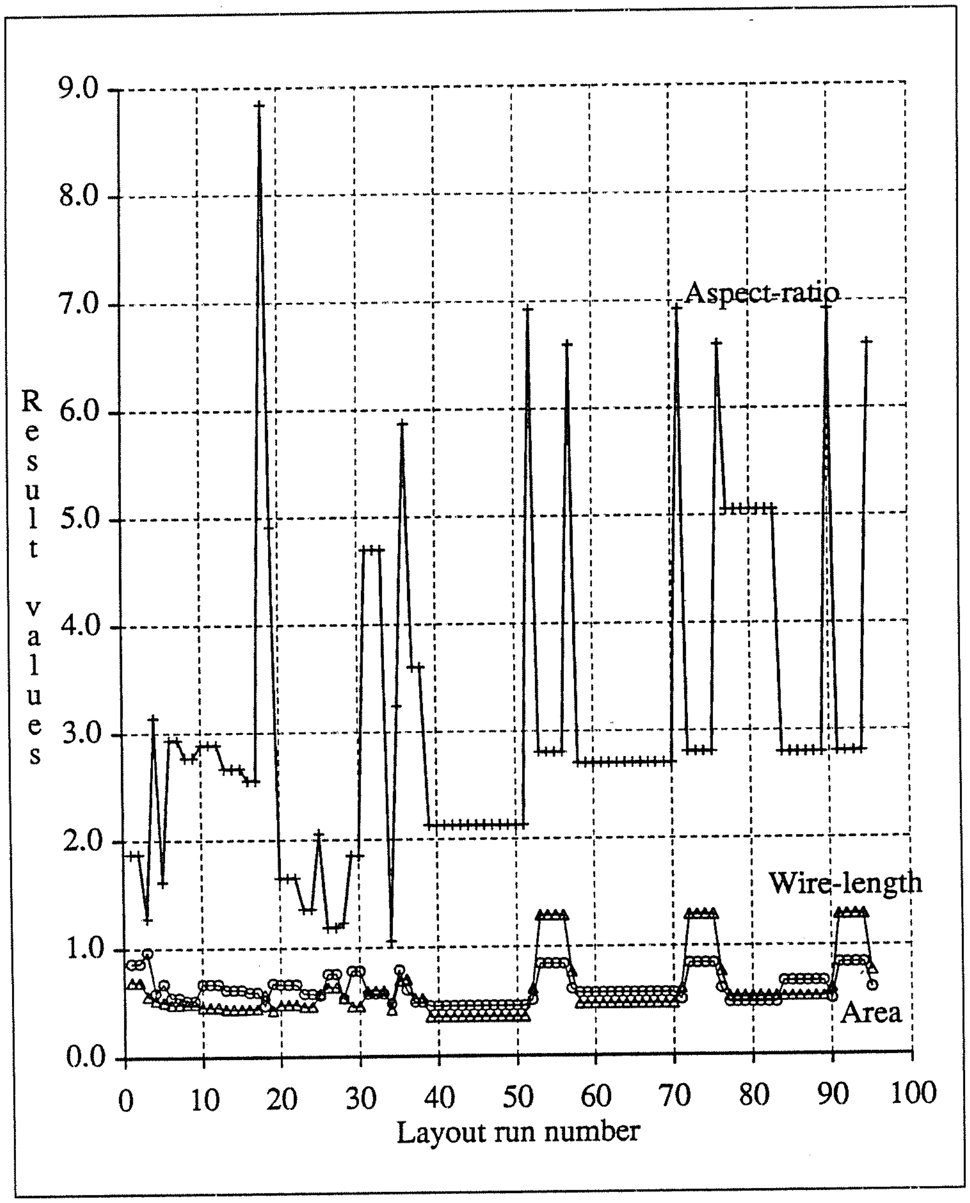


Figure 6.11: Recursion-threshold performance for layout *EX2*

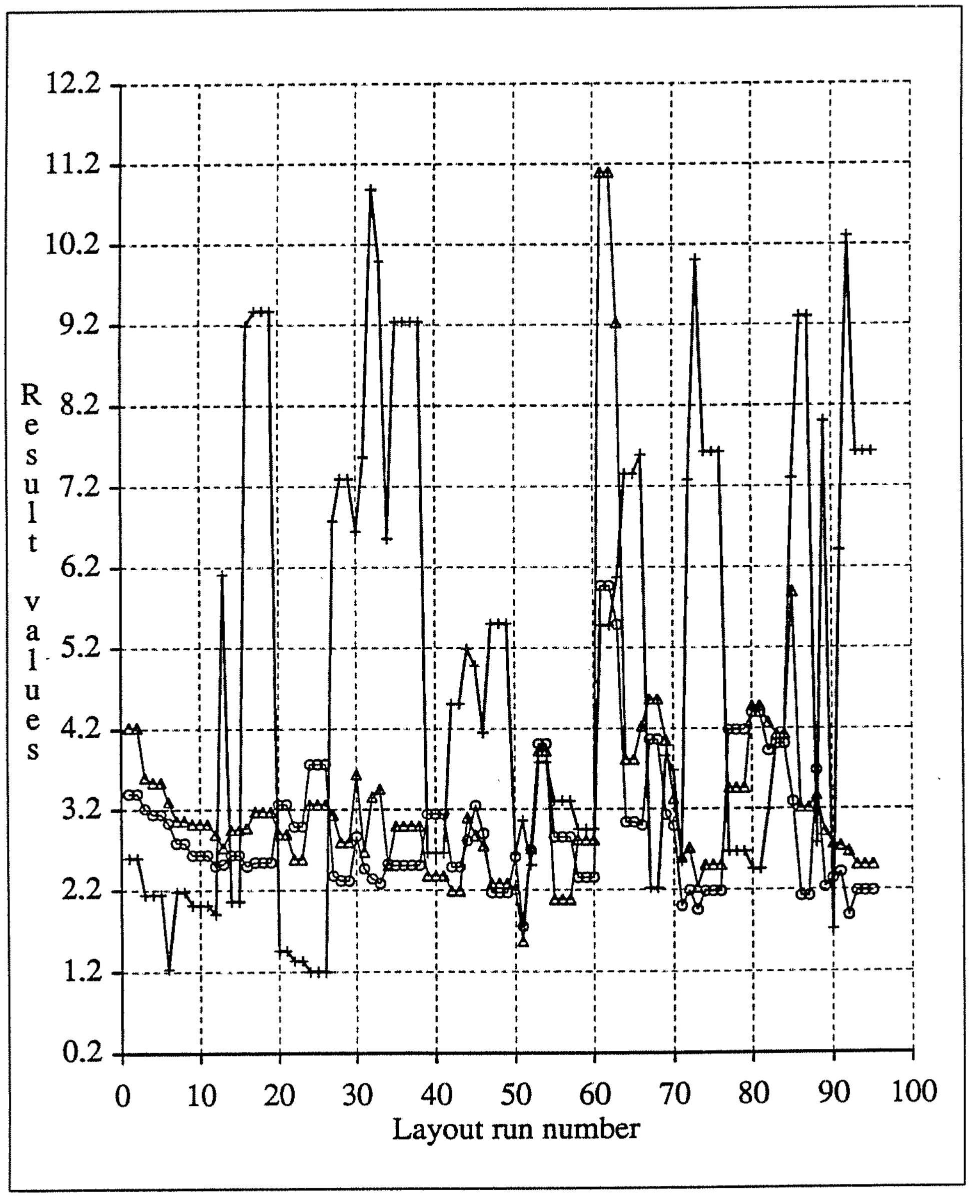


Figure 6.12: Recursion-threshold performance for layout *EX3*

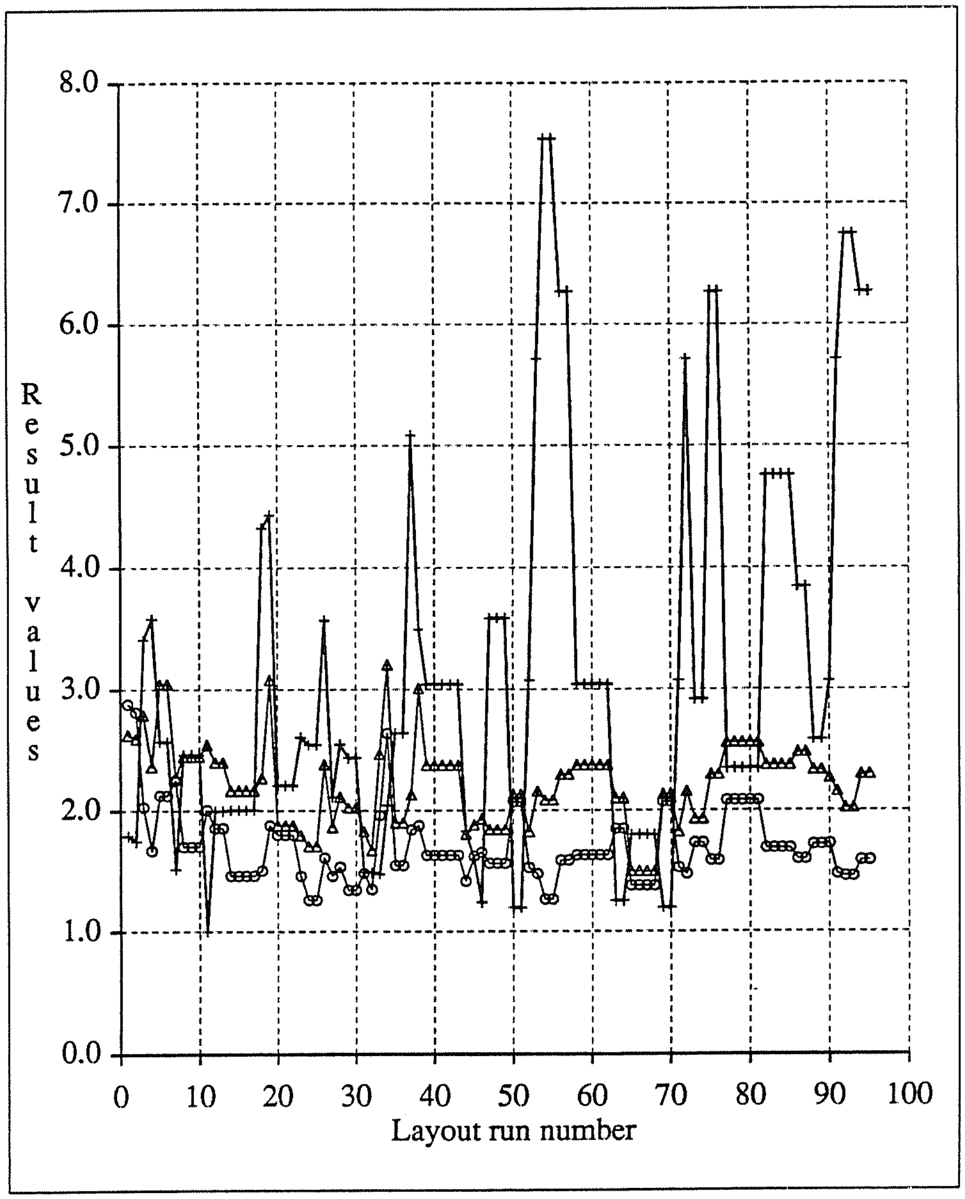


Figure 6.13: Recursion-threshold performance for layout *EX4*

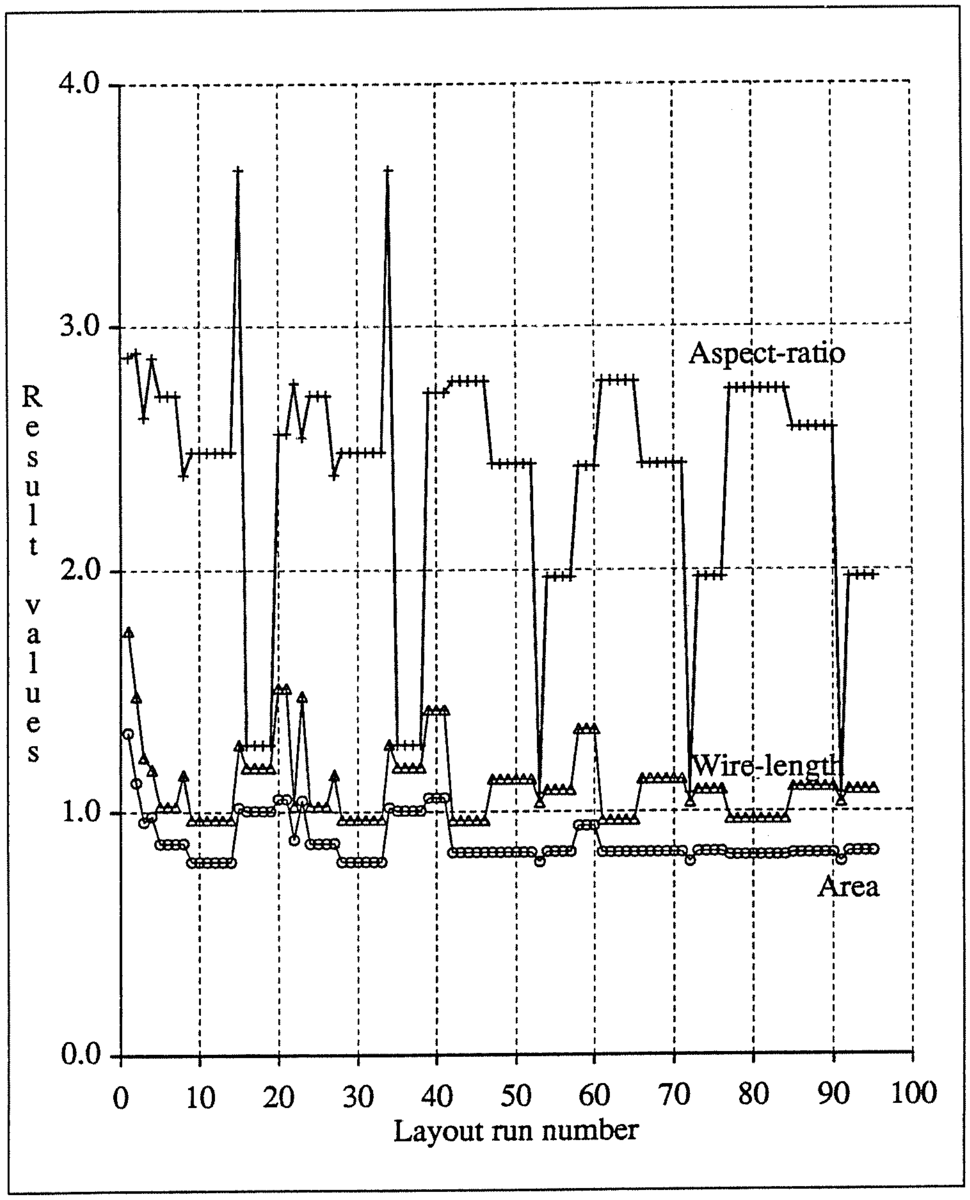


Figure 6.14: Recursion-threshold performance for layout *EX5*

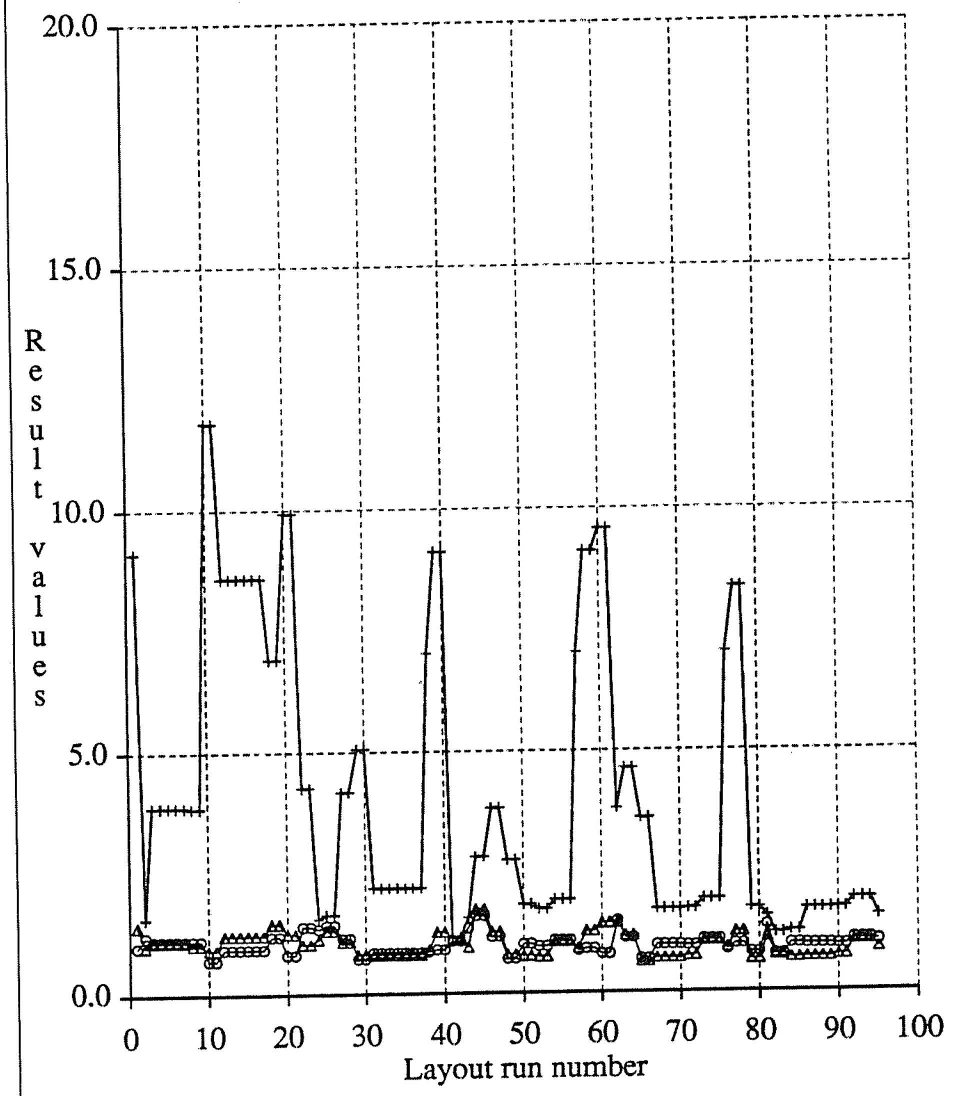


Figure 6.15: Recursion-threshold performance for layout *EX6*

sizes involved, which requires the function to evaluate and score a side relative to some other intrinsic characteristic of the layout itself. This would ensure that there would not be any significant variation in the weights needed for different layouts.

The SAT module increases its size by adding on more modules; hence, its size can only increase in units of the smallest edge length available in the module set, called the *min-size*. The min-size is therefore a natural choice of intrinsic measure, relative to which we construct the degree-of-fit function. This leaves only the exact form of the function to be decided. From the various layouts, both among the examples presented as well as other layouts we had studied, we noticed that a particular pattern was frequently repeated: that the distribution of module edges was not uniform. Rather, it was often the case that modules designed to fit together had identical, or near-identical, edge-lengths. Hence, we designed the degree-of-fit function to strongly bias those edges that were very nearly the size of the CORE edge under consideration. The degree-of-fit function was therefore built as the sum of two components, one being the actual ratio of the two dimensions, and the other being the bias that was added based on the size-difference between the edges. Since the dimension ratio is bounded by 1 (we always compute it as the ratio of the smaller dimension to the larger), this ensures that it will only come into play when the choice is between two modules, both of which have edge-sizes that are in the same general range. The degree-of-fit function was defined as:

degree-of-fit = (dimension ratio) + (bias value)

where bias value = 8 if SAT-size is within $\pm$ 5% of CORE-size

else, = 0 if SAT-size > CORE-size + min-size else, = 2 if SAT-size > CORE-size + (min-size ÷ 2)

else, = 4 if SAT-size > CORE-size + (min-size ÷ 4)

else, = 6 if SAT-size > CORE-size + (min-size ÷ 8)

else, = 3 if SAT-size < CORE-size – (min-size ÷ 2)

else, = 7 if SAT-size < CORE-size – min-size

else, = 5 if SAT-size < CORE-size – (min-size × 1.2)

else, = 3 if SAT-size < CORE-size – (min-size × 1.5)

else, = 1

The above function provides a means of evaluating the degree-of-fit of a candidate SAT-side relative to the CORE-edge and min-size, returning the score (a real number less than 10) as the degree-of-fit. This degree-of-fit value can now be added to the weighted SAT-connectivity value. Since only these two quantities contribute to the total for a side, we only need to vary one weight while keeping the other weight fixed, i.e.,

Score for a SAT module =

(SAT-connectivity × connectivity weight) + (degree-of-fit)

The Placer was accordingly tested at different weights for SAT-connectivity, keeping the weight for the degree of dimensional fit fixed at 1.

The table of layout numbers and SAT-connectivity weights is shown in Fig. 6.17, and the results from this series of tests are displayed in Figs. 6.18 — 6.23. As before, this testing was repeated for different <Recursion-threshold, Cluster-threshold> tuples in order to view the behavior in a more global setting; all other weights are fixed as before. Note that in all six layouts, variation in SAT-connectivity weight does not produce better layouts at other tuples than those corresponding to the best tuples determined for each layout in the previous tests. It is likely that, due to the extensive testing that has already been carried out, most of the weight values established are the best ones.

These results of the connectivity weight tests are quite surprising. With the exception of *EX3*, *all* layouts have their best behavior somewhere in the range of values of 20 — 30 for connectivity weight. *EX3* alone displays errant behavior, with its best layouts at weights in the range 1.2 — 5.0. However, it has a very close second-best placement at the weight range of 20 — 50. Taken all together, these results lead to the conclusion that a judicious combination of connectivity and dimensional fit yield the best criteria for satellite selection, rather than either connectivity or sizing alone, as has been the case in such approaches as min-cut or pure clustering. Also, another conclusion to be reached is that the balance point between connectivity and dimension considerations is fairly stable and less sensitive to differences in the layout problem, unlike the other factors tested so far. This is an encouraging discovery, since it means that with a high degree of confidence, we may use a weight of about 25 for most layouts. Finally, comparing the performance of the Placer with that of the original layouts, we note that for a greedy strategy, the results are very heartening indeed. Our results range from around 50% of the original area and wirelength (example *EX2*) as the most favorable, to around 130% in examples *EX3* and *EX4*.

Layout number	Weight	Cluster-threshold	Recursion-threshold
1	0.01		
2	0.2		
3	0.4		
4	0.6		
5	0.8		
6	1.0		
7	1.2		
8	1.4		
9	1.6		
10	1.8		
11	2.0		
12	2.4		
13	2.8	0.8	0.35
14	3.4		
15	4		
16	5		
17	7		
18	10		
19	15		
20	20		
21	25		
22	30		
23	40		
24	50		
25	1000		
26 — 50	0.01 — 1000	0.8	0.85
51 — 75	0.01 — 1000	2.0	0.75
76 — 100	0.01 — 1000	2.0	0.85

Figure 6.17: Connectivity weights vs. layout numbers in SAT-selection

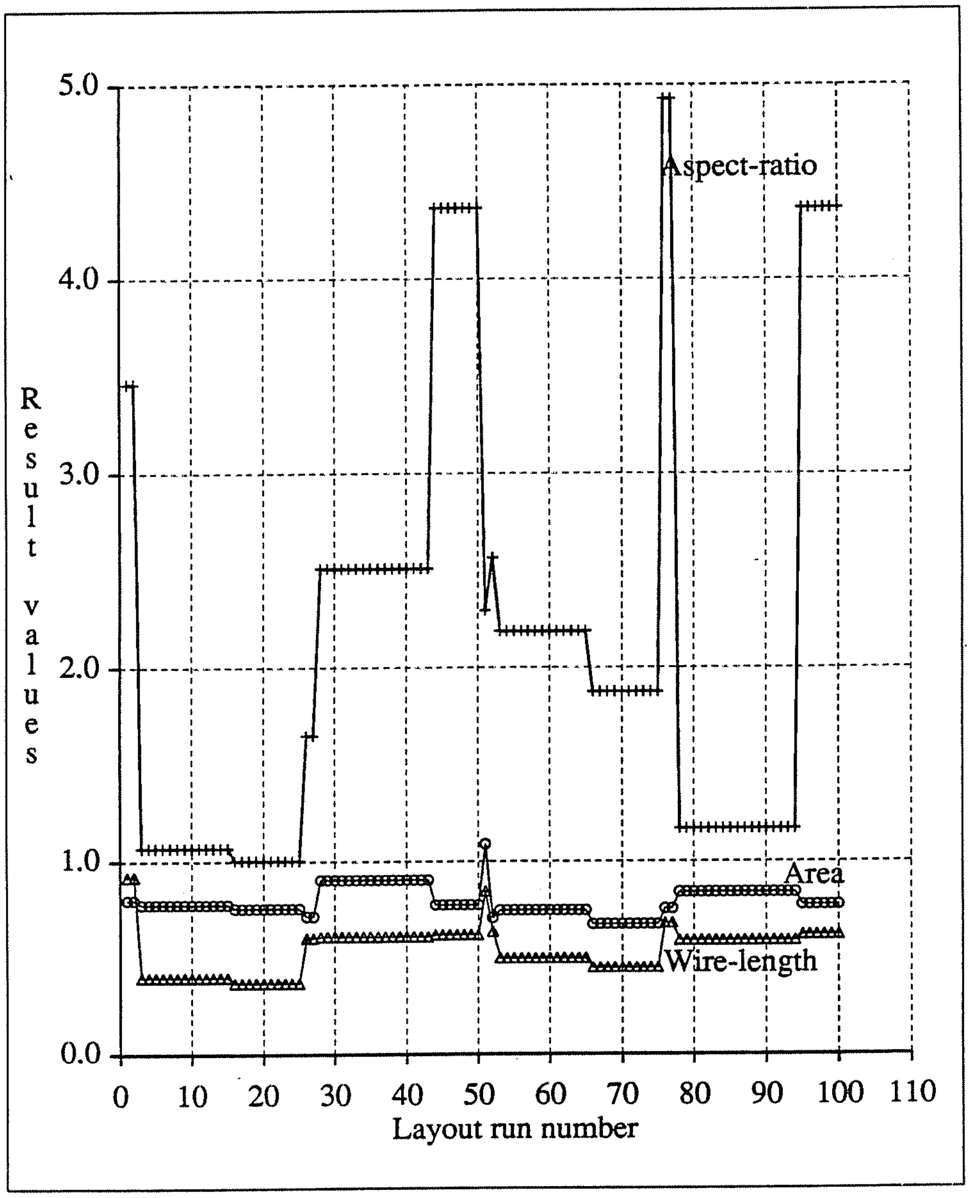


Figure 6.18: Satellite-connectivity performance for *EX1*

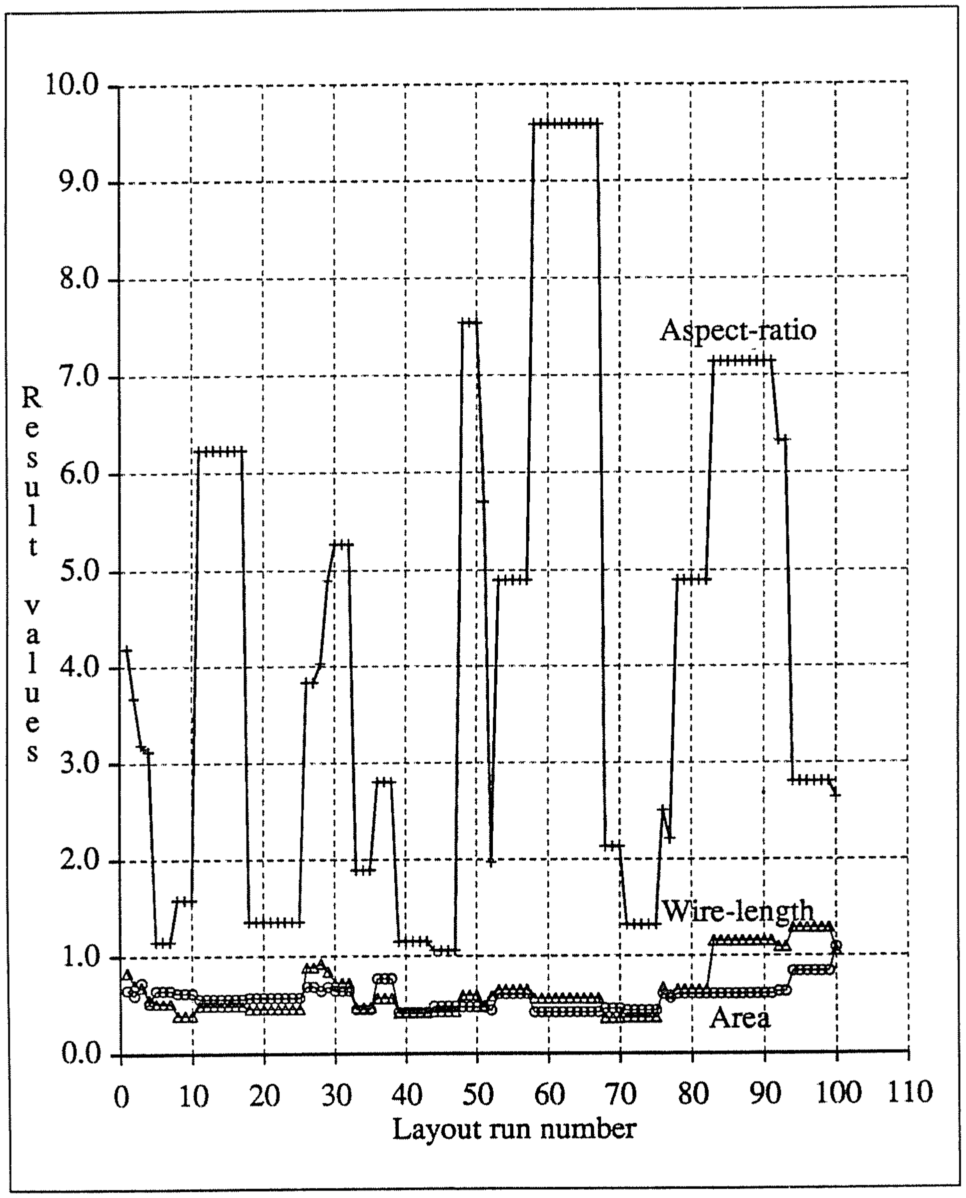


Figure 6.19: Satellite-connectivity performance for *EX2*

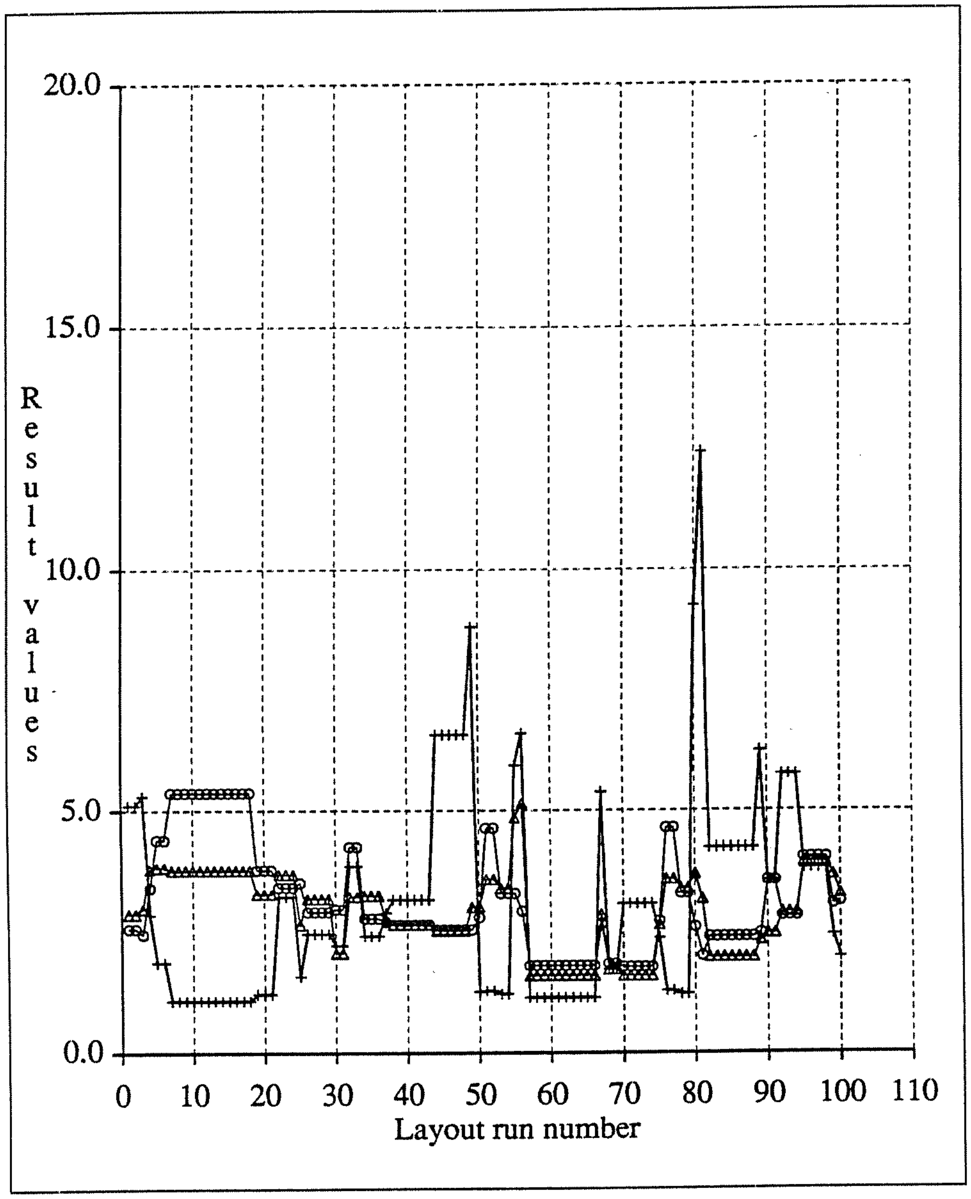


Figure 6.20: Satellite-connectivity performance for *EX3*

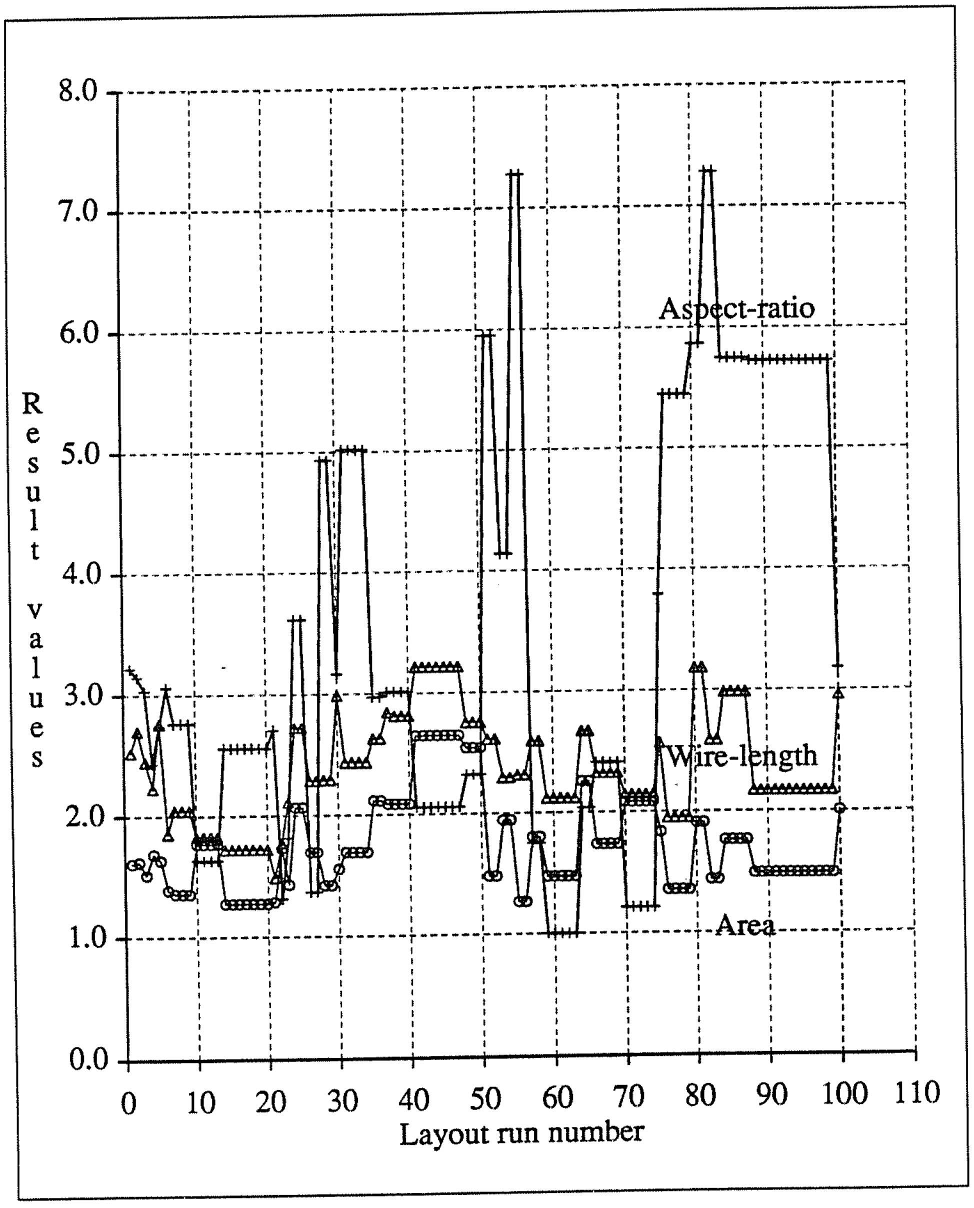


Figure 6.21: Satellite-connectivity performance for *EX4*

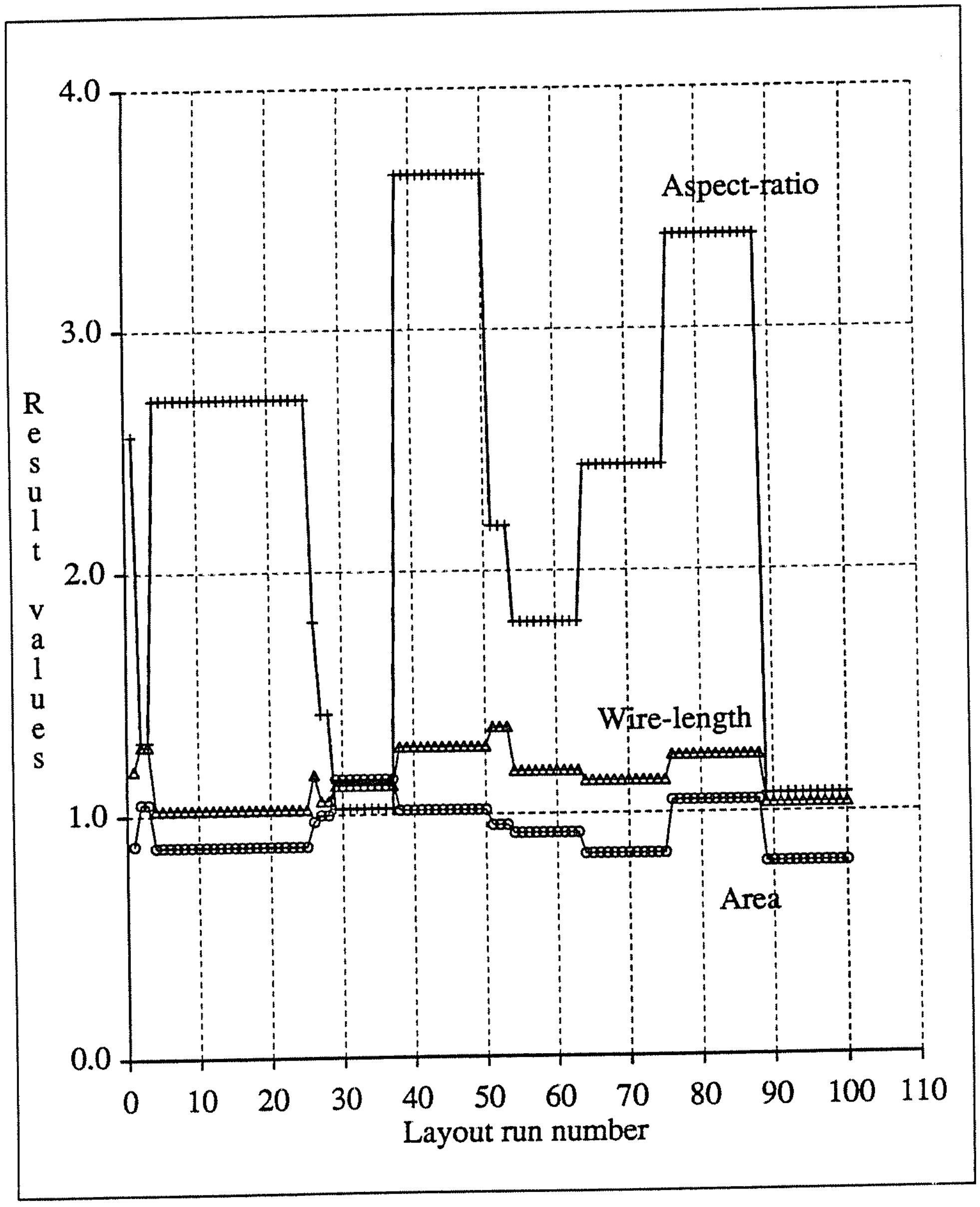


Figure 6.22: Satellite-connectivity performance for *EX5*

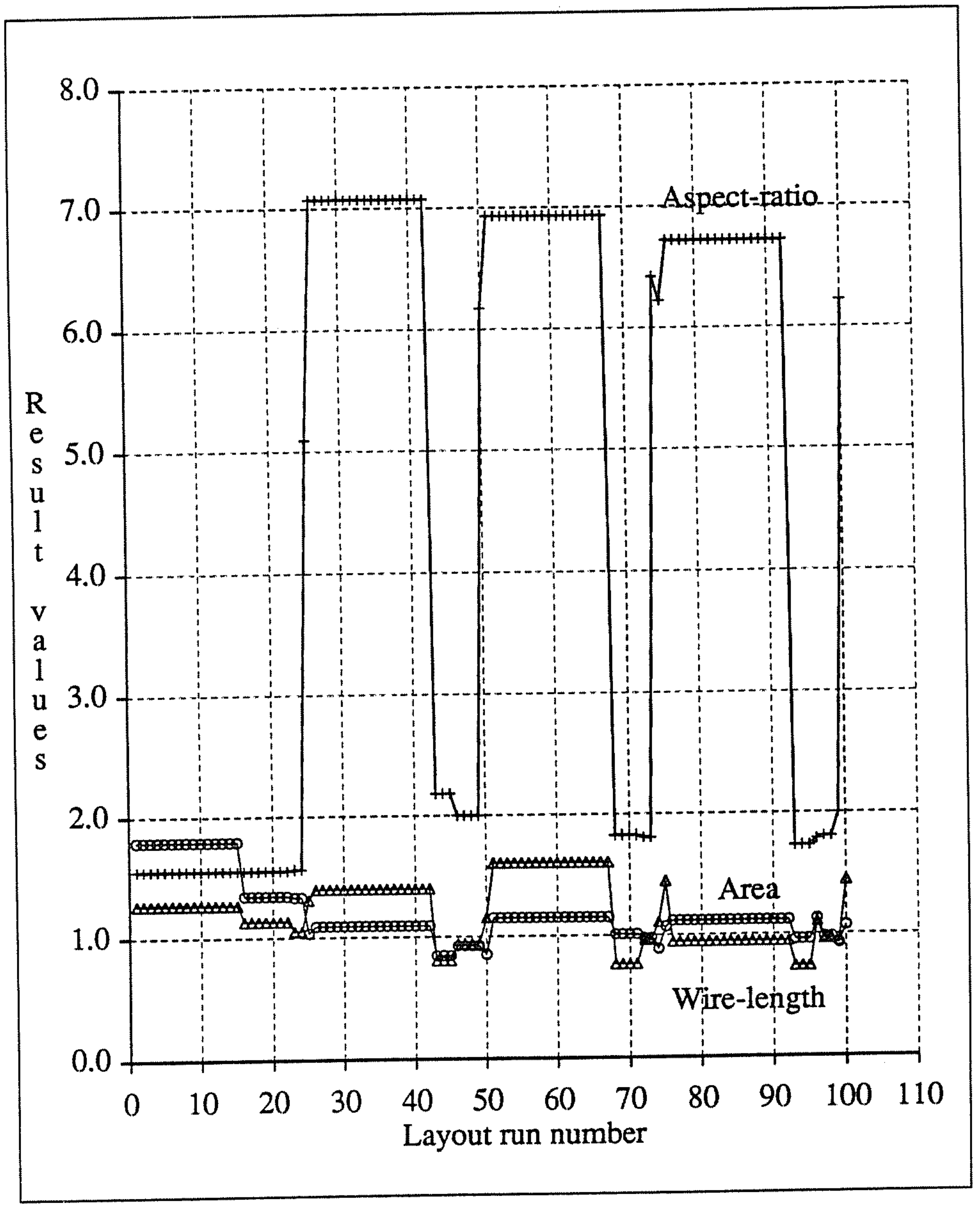


Figure 6.23: Satellite-connectivity performance for *EX6*

Layout name	Layout numbers	Weight	Cluster, Recursion threshold tuples
EX1	16 — 25	5 — ∞	0.8, 0.35
EX2	44 — 47	15 — 30	0.8, 0.85
EX3	57 — 66	1.2 — 5	2.0, 0.75
EX4	21	25	0.8, 0.35
EX5	89 — 100	3.4 — ∞	2.0, 0.85
EX6	43 — 49	10 — 50	0.8, 0.85

Figure 6.24: Summary of Connectivity weights in SAT selection

6.5. Side-connectivity: making orientation decisions

We have discussed in detail the behavior of the Placer in the context of clustering, recursion and satellite-selection decisions. These criteria are all intricately woven together, with decisions at some levels affecting the outcome at other levels. For instance, it was seen that a modification in the recursion threshold led to a change in the best cluster-threshold value for example *EX1*. Although the same is generally true of the module orientation criteria, we have insulated the actual orientation decision from the satellite-selection decision by making global orientation decisions rather than greedy ones. Specifically, when a SAT module is to be chosen, orientation decisions are made for *all* the candidate SAT modules before allowing the satellite-selection procedure to continue. A modification in the orientation criteria directly affects the satellite selection; but once SAT has been chosen, the orientation decision will not run counter to the choice.

The fact that the orientation decision is a major input to the SAT selection process suggests that testing of the orientation criteria should also be carried out in a global framework, and not locally within the best pre-determined values of the weights from the preceding sections. Accordingly, we present the results from testing of the orientation criteria over a series of tuples of <Recursion-threshold, Cluster-threshold, Connectivity-weight>.

To make an orientation decision, a "score" is again generated for *each side* of a candidate SAT module. The inputs for determining this score are: connectivity between the CORE module and the net-lists of the four sides of the SAT module, and lengths of the edges of the SAT module. The conversion of a dimension into an appropriate quantity for comparison with or combination with other quantities has been discussed at length in the previous section; the same degree-of-fit function is used here. The connectivity that we consider now is not between modules, but between CORE and the *sides* of SAT; hence, it is handled in a different way. The orientation of SAT determines the length of wire-routing that will be necessary, not only from the side of SAT that is chosen to face CORE, but also from the remaining three sides of SAT. The side-connectivity measure therefore needs to take the overall wiring length into consideration. This is approximated as a weighted sum of the number of nets on each of the three sides nearest to CORE, that are connected to CORE. Figure 6.25 illustrates how this side-connectivity value is obtained. Given a known side of CORE that will face SAT, we calculate the side-connectivity value for a particular side S of SAT as the sum of the nets on side S, plus half the nets on the two sides of SAT adjacent to S, considering only those nets that are connected to CORE. In Fig. 6.25, S is the left side of SAT, and X, Y and Z are the nets from the

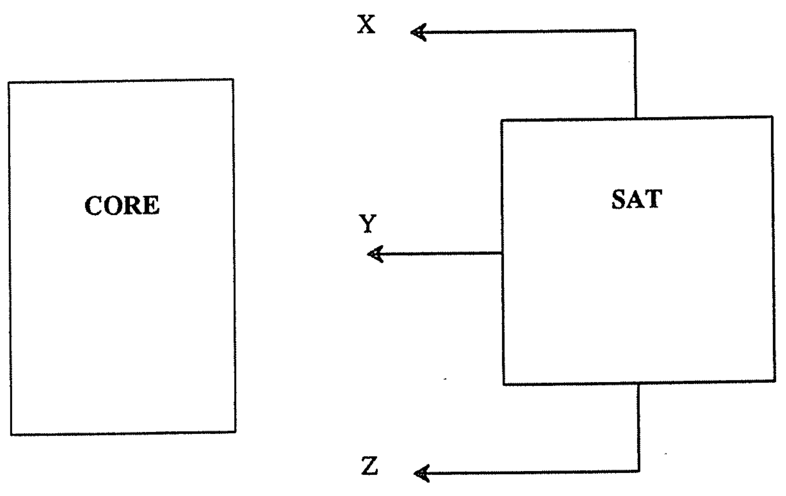


Figure 6.25: Side-connectivity computation

top, left and bottom sides of SAT that are connected to CORE. The side-connectivity computed for the left side of SAT is, therefore,

$$Y + (X + Z) \div 2$$

This side-connectivity is then converted as before by dividing it by the average nets per module. The idea is to weight nets on a facing side higher than nets on adjacent sides, and to not count nets on the opposite side at all. This side-connectivity is computed for each of SAT's four sides, as is the degree-of-fit. The side of SAT with the highest total for degree-of-fit + (side-connectivity $\times$ side-connectivity weight)

is selected as the best side to face CORE, since this side would have the strongest combination of high connectivity close to the CORE, as well as a good dimensional match.

As before, we begin with the table of Fig. 6.26, which displays the series of layout numbers and their corresponding weight values. Figures 6.27 — 6.32 detail the behavior of the examples over the range of side-selection connectivity weights, with the summary of our conclusions tabulated in Fig. 6.33. Two points emerge from this test. The first is that, with the exception of example *EX6*, all layouts do well at a weight in the region of 0.8 for connectivity in side-selection. Hence, it is again a fair conclusion that orientation decisions can be successfully based on both connectivity and on the dimensions of the edges of the modules involved. The knowledge that we gain from these tests is the information that greedy satellite-selection can yield performance comparable to more complex global approaches: we should point out again that of the examples, only *EX3* and *EX4* are placed with larger final areas than the originals. The second point to note is that a changed value for side-selection connectivity weight (from the pre-determined value used in previous tests) leads to better layouts for two of the examples, *EX4* and *EX6*.

Layout number	Weight	Cluster, Recursion, SAT-connectivity weight tuples
1	0.01	0.8, 0.35, 25
2	0.2	
3	0.4	
4	0.5	
5	0.6	
6	0.7	
7	0.8	
8	0.9	
9	1.0	
10	1.2	
11	1.4	
12	1.7	
13	2.0	
14	2.3	
15	2.6	
16	3	
17	4	
18	5	
19	10	
20	100	
21 — 40	0.01 — 100	0.8, 0.85, 25
41 — 60	0.01 — 100	2.0, 0.75, 3
61 — 80	0.01 — 100	2.0, 0.85, 25

Figure 6.26: Side-connectivity weights vs. layout numbers

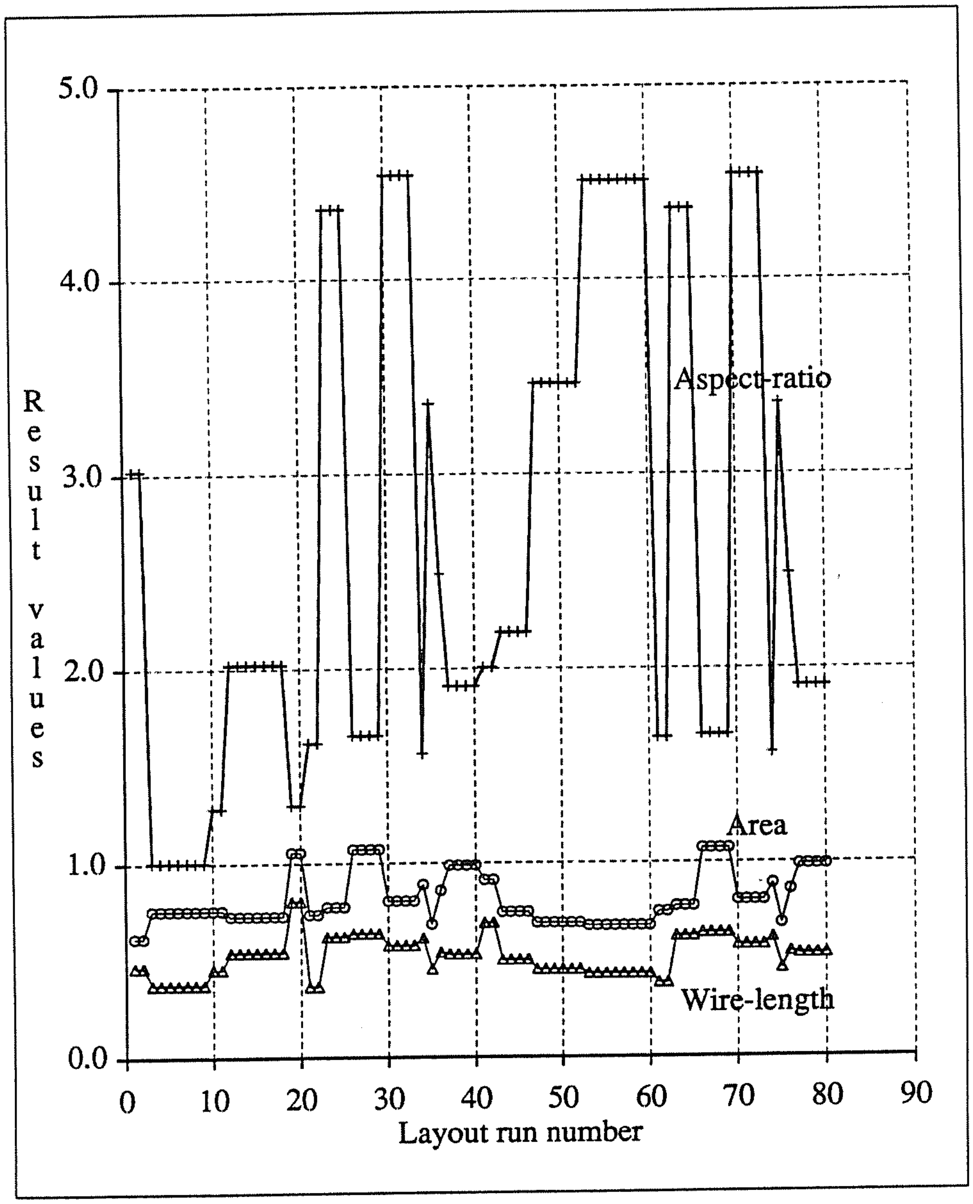


Figure 6.27: Connectivity weight in SAT selection for *EX1*

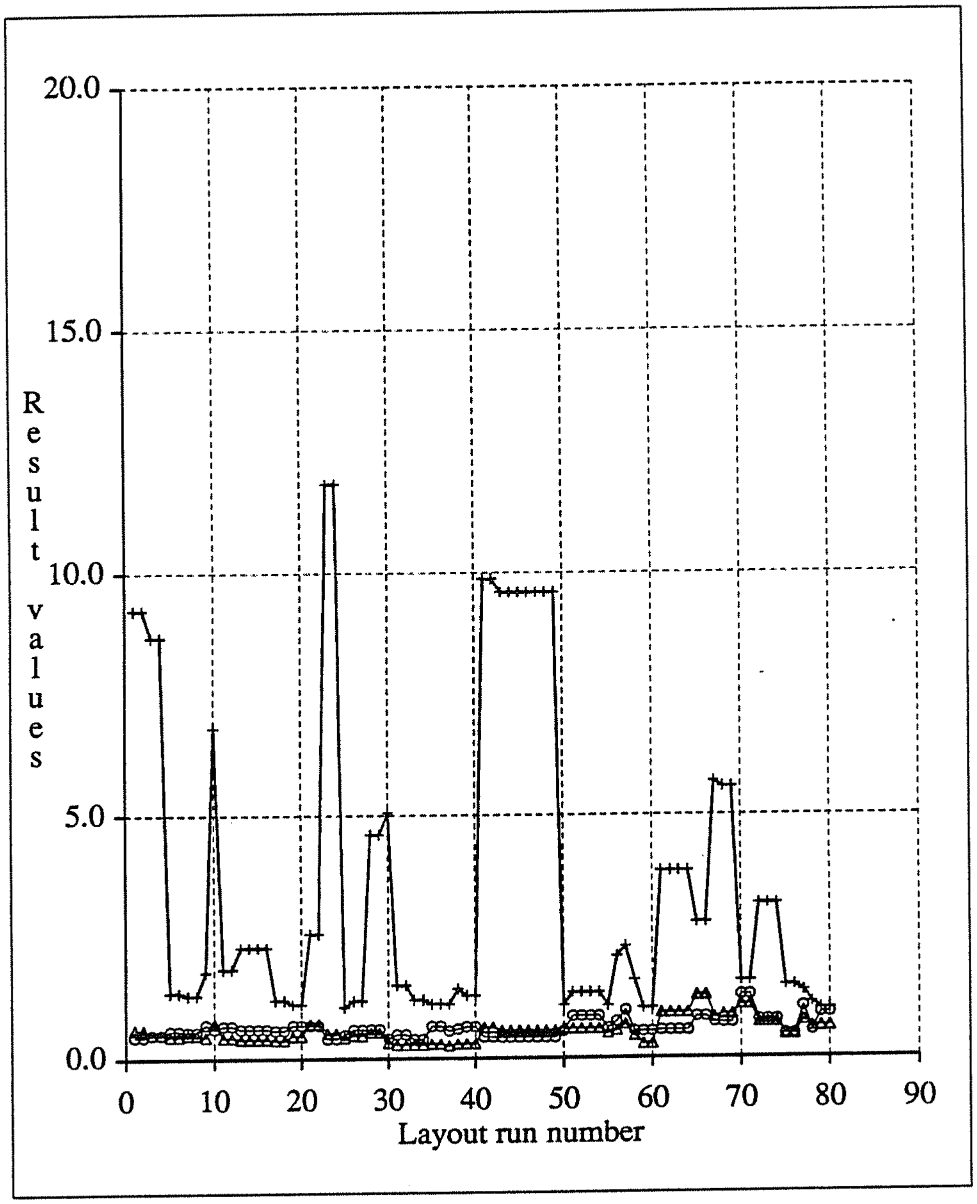


Figure 6.28: Connectivity weight in SAT selection for *EX2*

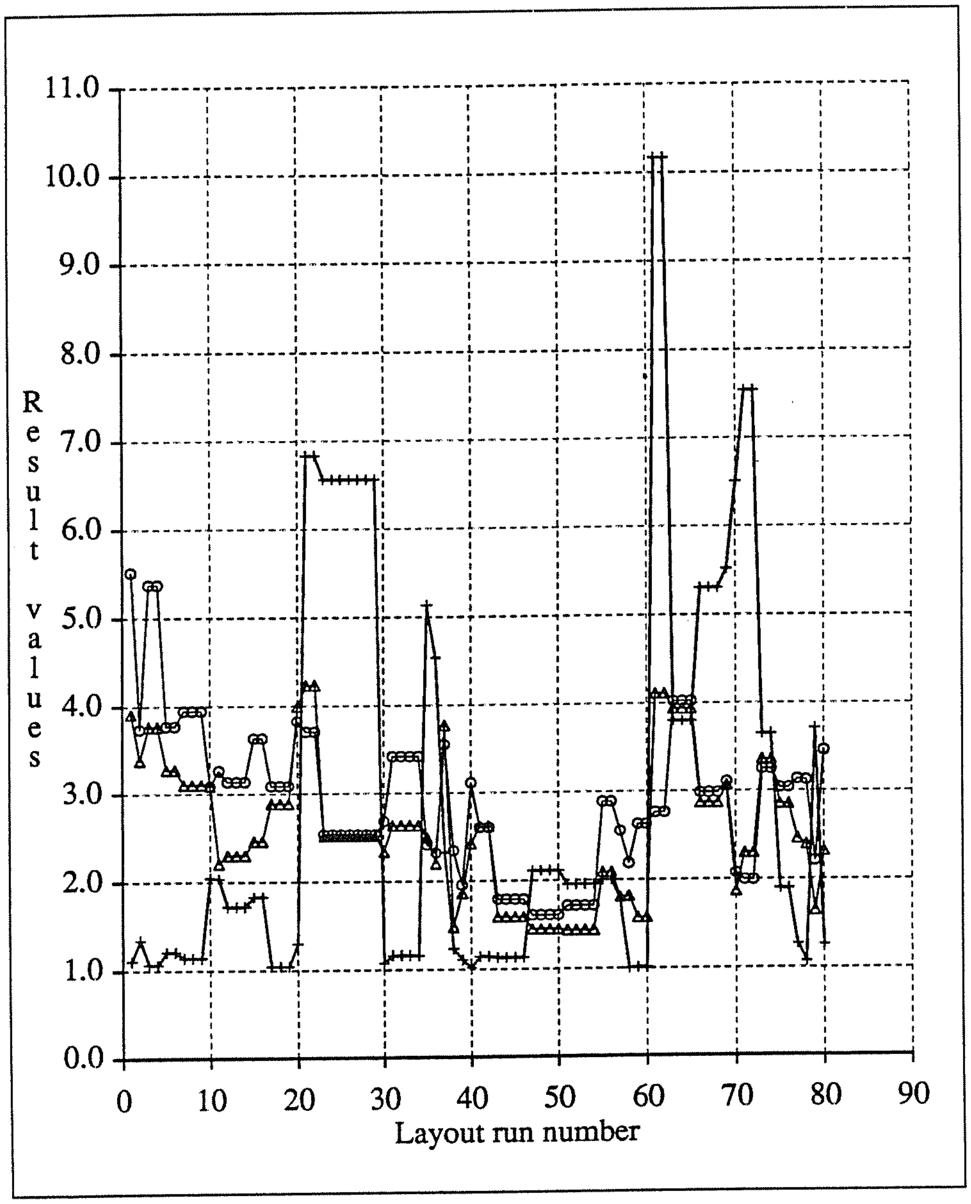


Figure 6.29: Connectivity weight in SAT selection for *EX3*

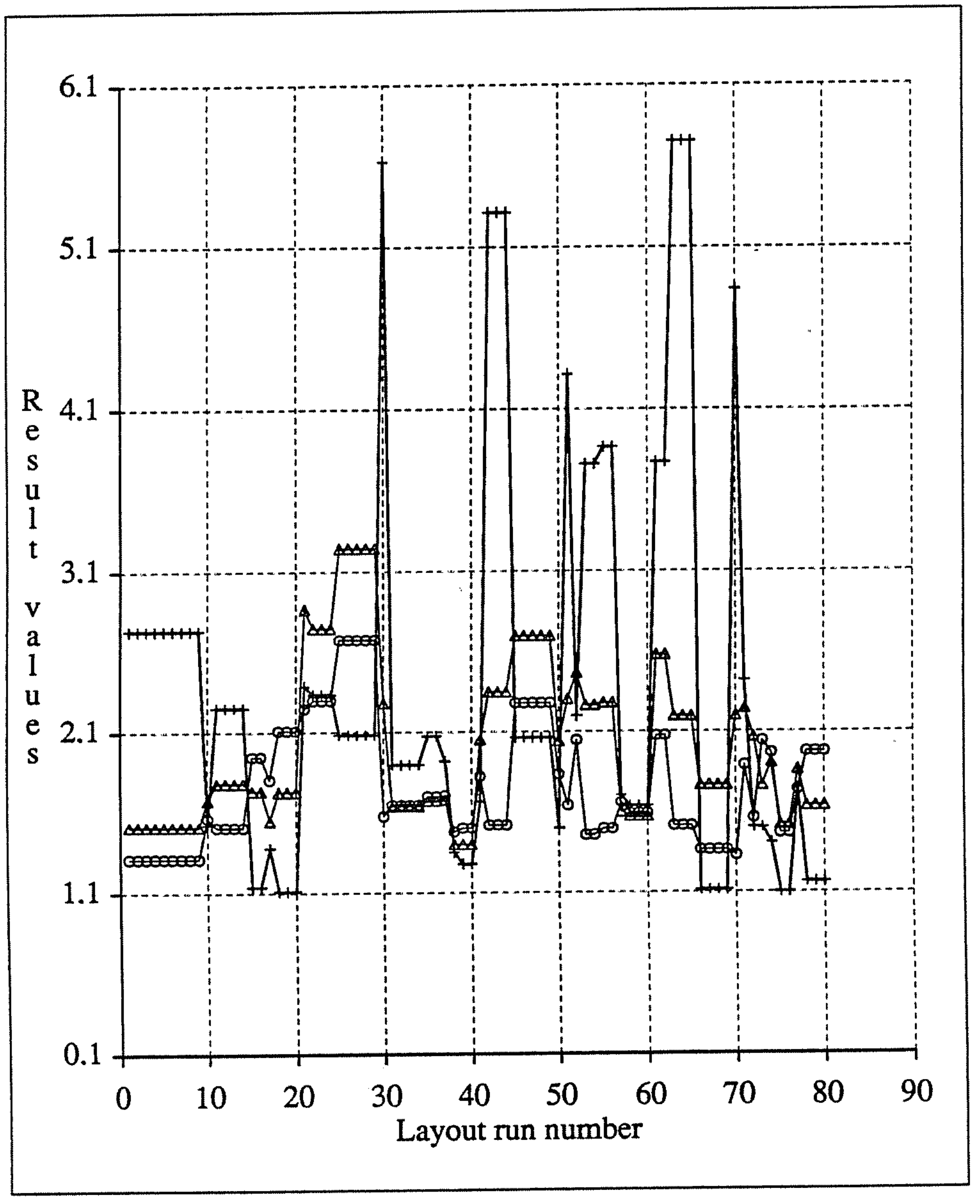


Figure 6.30: Connectivity weight in SAT selection for *EX4*

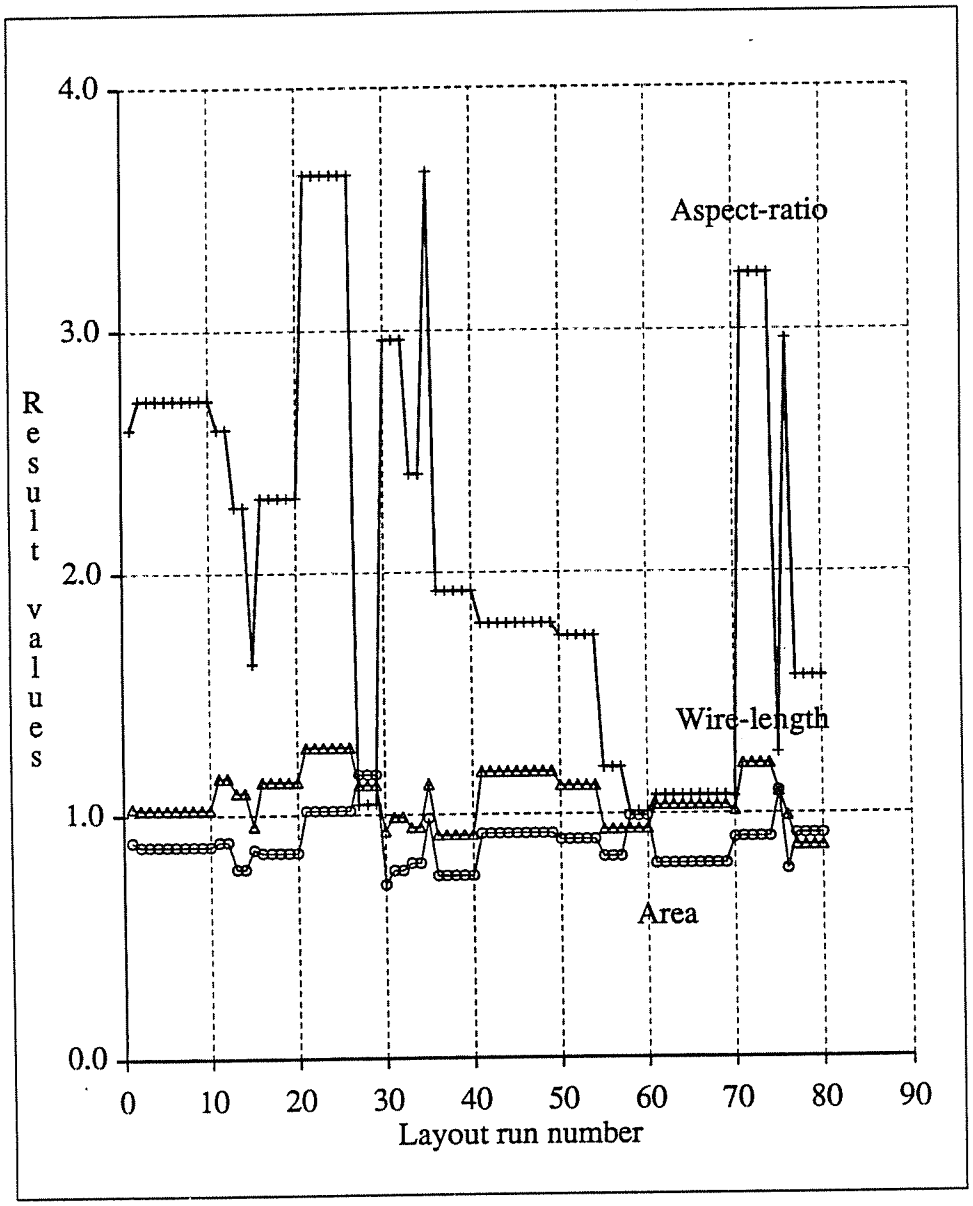


Figure 6.31: Connectivity weight in SAT selection for *EX5*

(number of channel nets not connected to both CORE and SAT)

In this series of tests, the best subsets of tests have been selected to show only the relevant information, since it was noted that variation in the penalty weight did not reveal any further changes in the best tuples of weights for any of the layouts. The

Layout number	Weight	Layout number (contd.)	Weight (contd.)
1	0.01	16	1.8
2	0.1	17	2.0
3	0.3	18	2.2
4	0.5	19	2.4
5	0.6	20	2.8
6	0.7	21	3.2
7	0.8	22	3.6
8	0.9	23	4.0
9	1.0	24	4.5
10	1.1	25	5
11	1.2	26	6
12	1.3	27	7
13	1.4	28	8
14	1.5	29	10
15	1.6	30	100

Figure 6.34: Penalty weights for non-channel nets vs. layout numbers

penalty-factor behavior is very disciplined, uniformly producing the best layouts at a range of values of 0.7 — 0.9 for all the examples. Once again, the fact that this weight is an intermediate value, and not an extreme value (such as near-zero or near-∞) supports the the idea of imposing a penalty on certain orientations on the basis of "unconnectivity".

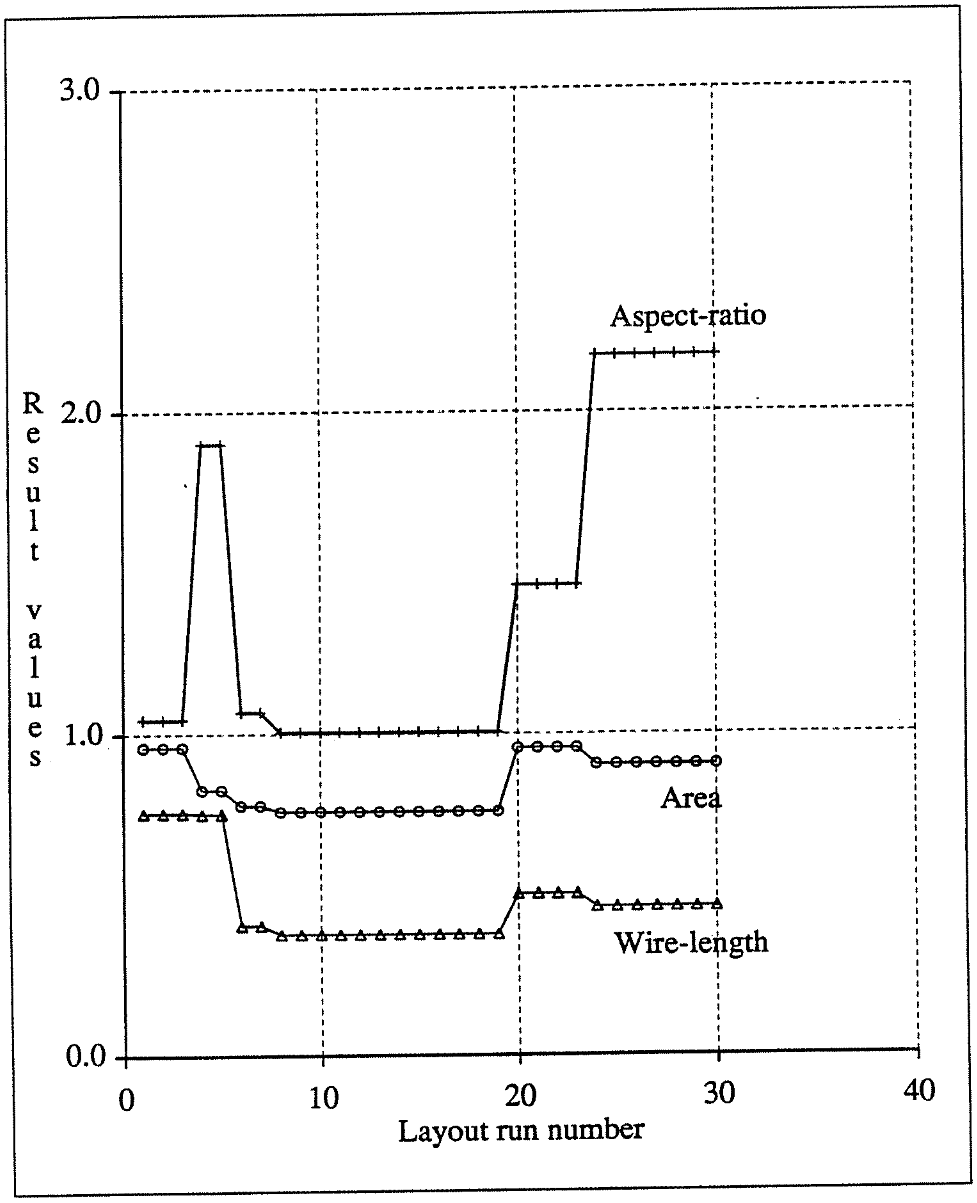


Figure 6.35: Side-penalty performance for *EX1*

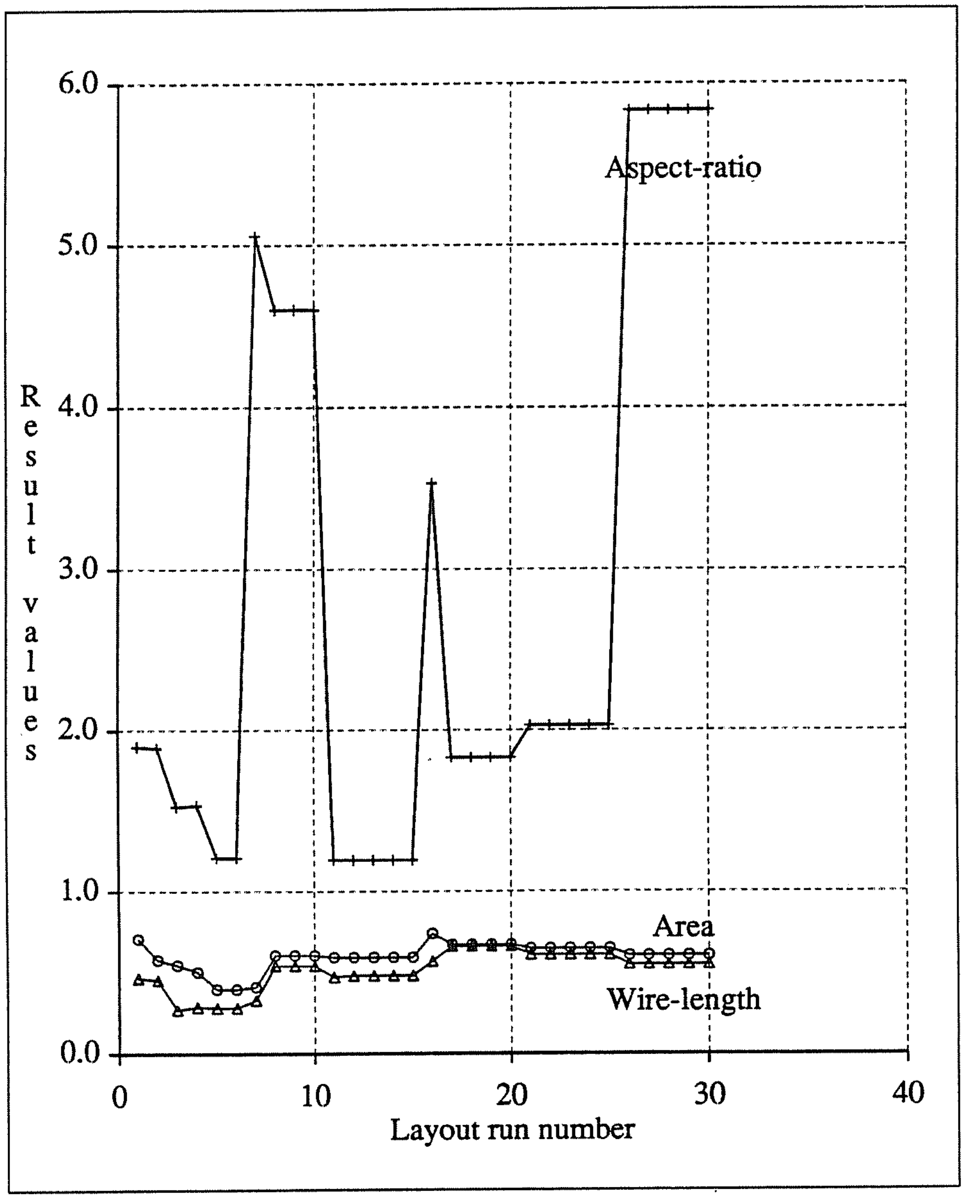


Figure 6.36: Side-penalty performance for *EX2*

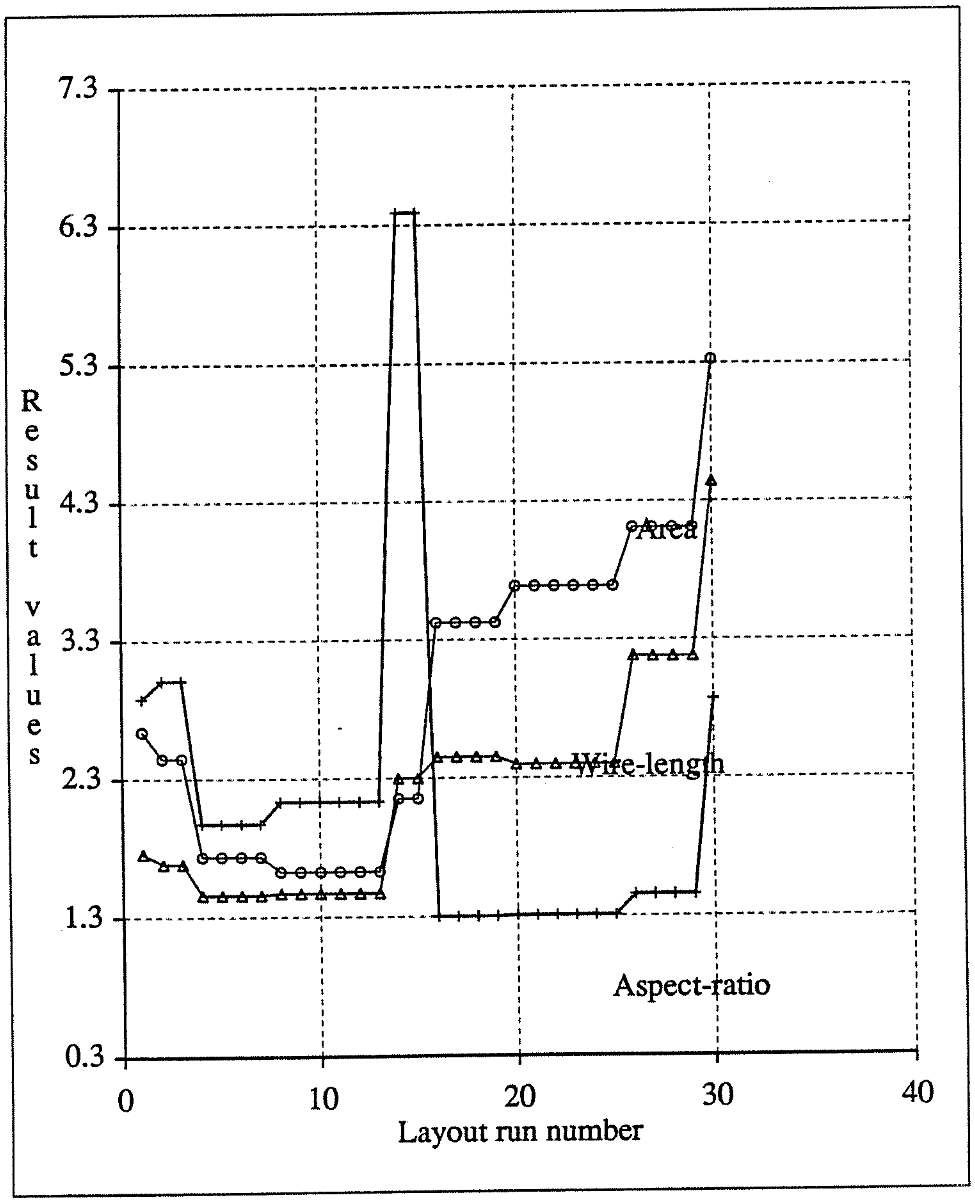


Figure 6.37: Side-penalty performance for *EX3*

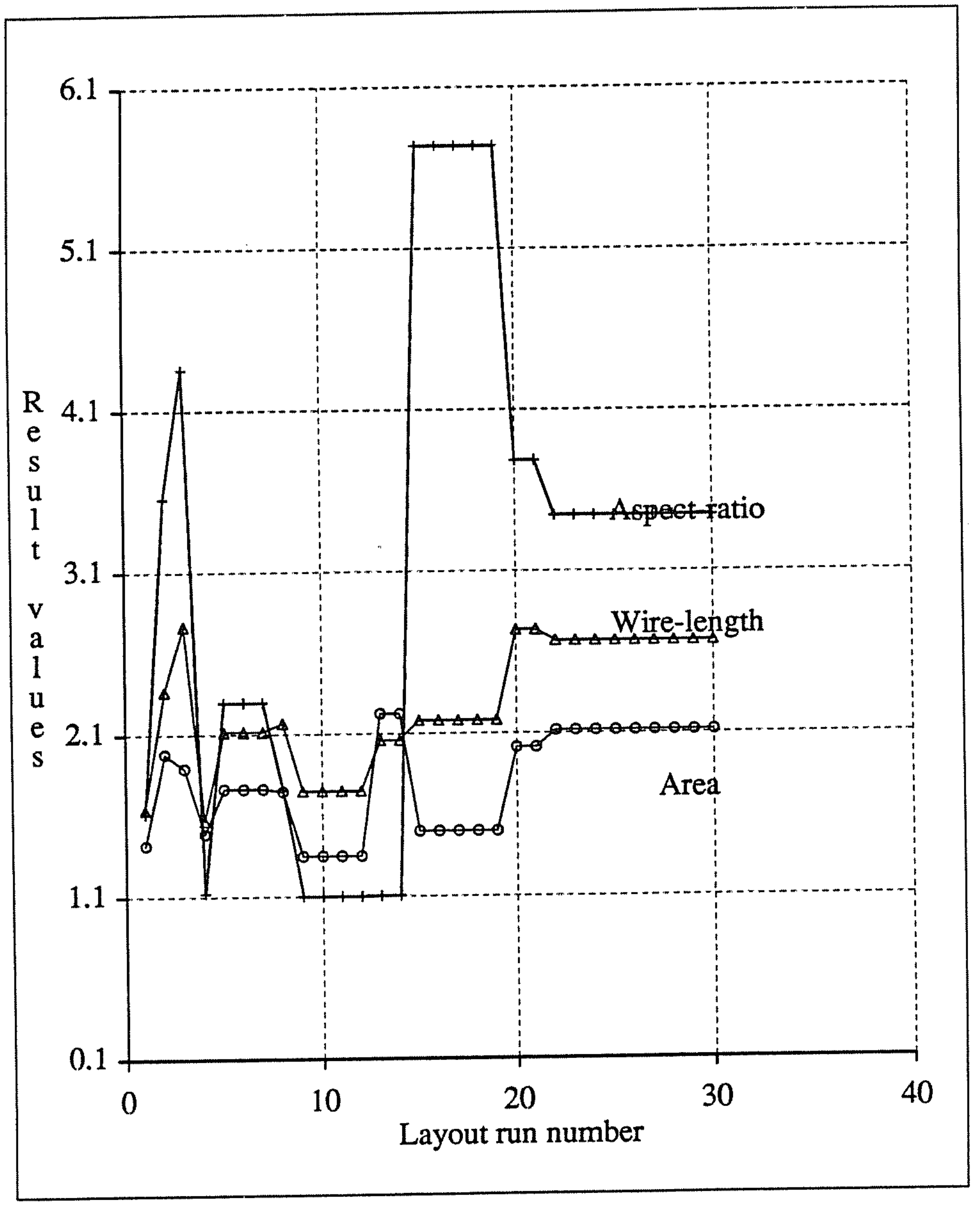


Figure 6.38: Side-penalty performance for *EX4*

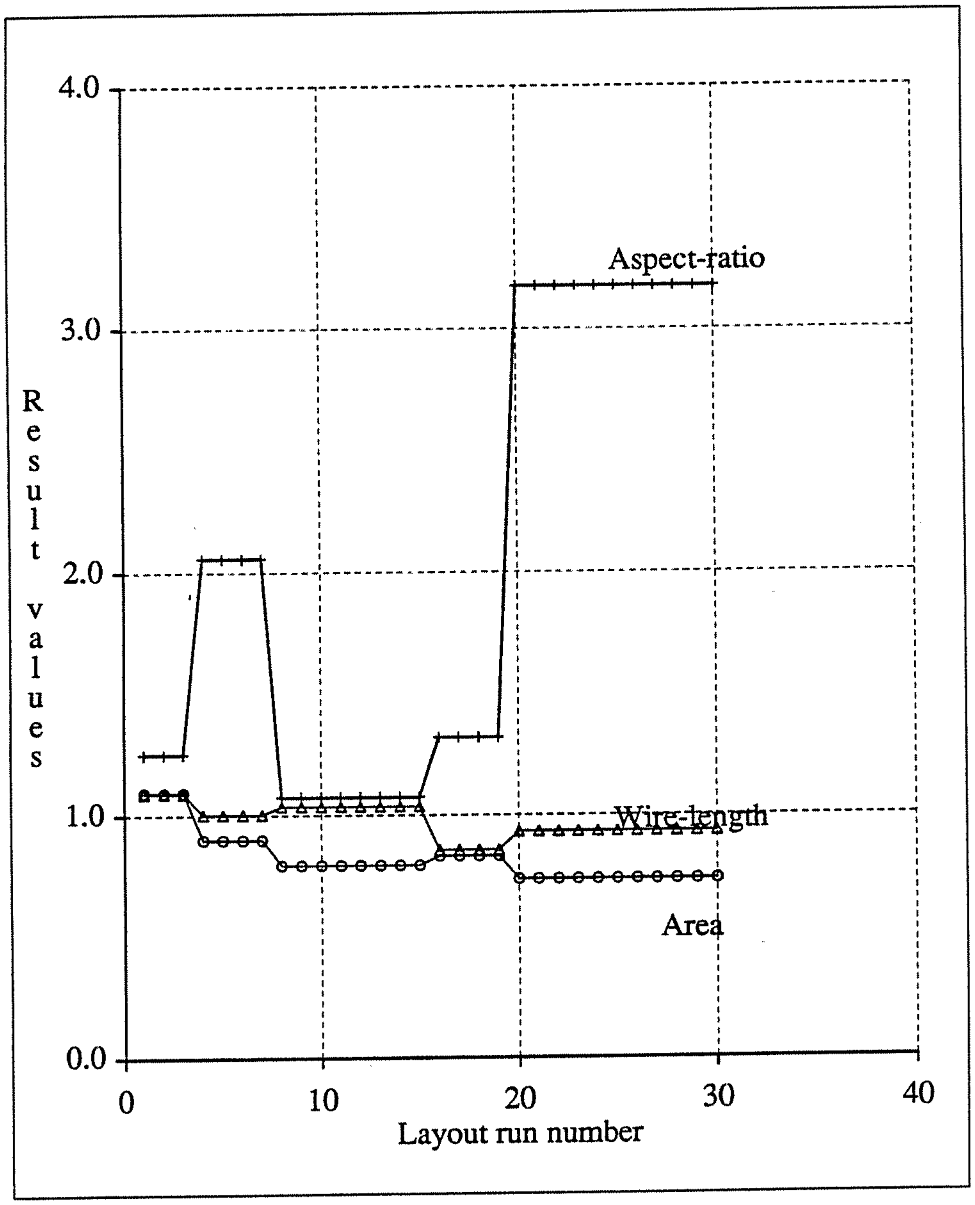


Figure 6.39: Side-penalty performance for *EX5*

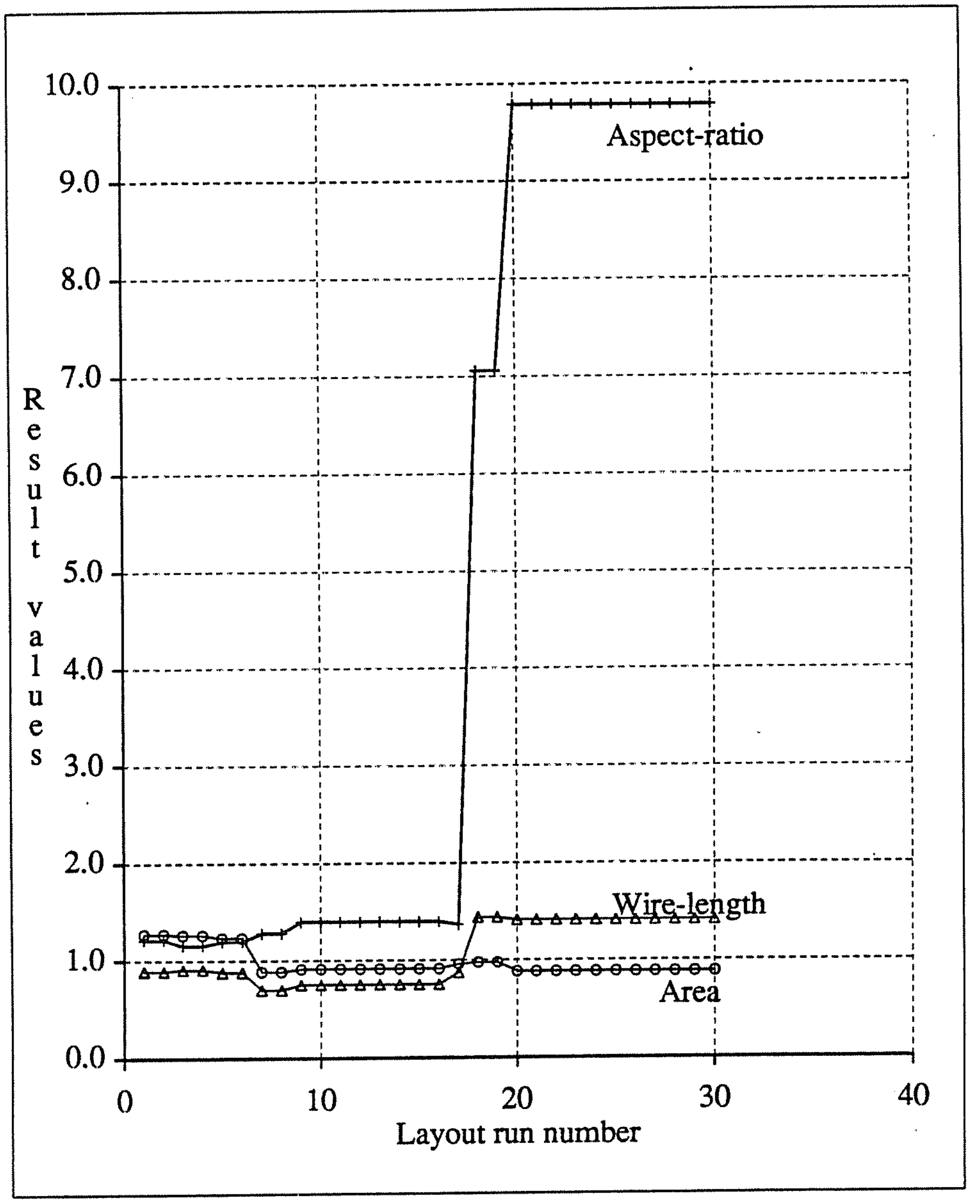


Figure 6.40: Side-penalty performance for *EX6*

Layout name	Layout numbers	Penalty weights for non-channel nets	Cluster, Recursion, SAT-connectivity, Side-connectivity weights
EX1	8 — 19	0.9 — 2.4	0.8, 0.35, 25, 0.8
EX2	5 — 6	0.6 — 0.7	0.8, 0.85, 25, 0.8
EX3	8 — 13	0.9 — 1.4	2.0, 0.75, 3, 0.8
EX4	8 — 11	0.9 — 1.2	2.0, 0.85, 25, 0.8
EX5	8 — 15	0.9 — 1.6	2.0, 0.85, 25, 0.8
EX6	7 — 8	0.8 — 0.9	0.8, 0.35, 25, 2.5

Figure 6.41: Summary of penalty weights in orientation

6.6. Reduction in weights between clustering cycles

We conclude this presentation of results by discussing results for the fine-tuning of the weight factors between clustering cycles. In section 4.3, the recursive building process was described; in that process, whenever all the modules in a set of clusters have been built up into supermodules, the Closest Neighbor Graph is rebuilt for the new set of modules. At this point, it may be possible to effect some improvement in the performance of the Placer by reducing some of the weight factors. For instance, as supermodules grow, the number of nets per module will clearly decrease. Therefore, it may be a good idea to modify some of the factors dealt with in previous sections so that after each round of clustering, the factor is multiplied by a corresponding reduction factor.

Two sets of tests were run for this purpose, one dealing with the cluster-threshold weight, and the other with connectivity weight for SAT selection. The results are summarized in Fig. 6.42; since this section of the results is more of a detail than a fundamental concept, the graphs are omitted. We merely point out the uniform behavior of all examples for both the reduction factors considered. In the case of the threshold cluster-membership connectivity, a reduction factor of 0.5 is seen to be within the best value range of 5 out of the 6 examples. *EX4* deviates from this, requiring a reduction factor of at least 1.0, i.e., an *increasing*, or at least, a non-decreasing cluster threshold. For the connectivity weight for SAT selection, all layouts uniformly do well over a range of reduction of 0.4 — 0.8, indicating that a choice of 0.6 for the

Layout name	**Cluster-threshold reduction factors**	**SAT-connectivity reduction factors**
EX1	0.4 — 0.8	0.2 — ∞
EX2	0.4 — 0.6	0.2 — ∞
EX3	0.4 — 0.5	0.3 — 1.1
EX4	1.0 — ∞	0.4 — 0.8
EX5	0.5 — ∞	0.2 — ∞
EX6	0.5 — 0.8	0.4 — 1.2

Figure 6.42: Summary of weight-reduction factor performance

SAT-connectivity reduction factor would satisfactorily cover all cases.

6.7. Overall trends in the successful weight values

We have finally reached the stage where we can step above the detailed setting of weight values, and discuss the overall picture: what knowledge has been gained regarding the appropriate weights to use when dealing with a module set to be placed and routed. We have mentioned one of the benefits of a greedy approach as the high speed with which a layout can be produced. This benefit is, of course, contingent on our identifying weights that need to be used. If this cannot be done, then the greedy approach loses its edge, since producing a layout would mean iterating through many different layouts for different sets of weights. Although individual layouts may be speedily completed, obtaining a *good* layout would be reduced to a long search through many possible candidate layouts, with less compelling reasons to choose this approach over any of the global approaches discussed in chapter 2.

The results for all six layouts have been tabulated, along with additional information about the examples themselves, in Fig. 6.43. We should again point out two facts. Firstly, of these six layouts, *EX3* was generated by a floor-planner, which involves human interaction in producing the design of the system; and *EX4*, *EX5* and *EX6* were compacted by an additional stage after the placement and global routing stages. Secondly, the Placer overestimates its channel sizes. For instance, the layout produced for *EX1* was hand-routed, and it was seen that *every single channel was larger than required.* It would appear that our channel size-estimation has been pessimistic, and therefore, an additional improvement could be expected from including a

	EX1	*EX2*	*EX3*	*EX4*	*EX5*	*EX6*
Number of modules	7	18	17	19	8	16
Number of nets	34	37	89	27	17	79
Smallest edge[3]	144	144	84	84	72	62
Largest edge	492	468	780	516	624	391
Smallest # of nets on 1 module	2	2	2	2	2	5
Largest # of nets on 1 module	15	6	34	8	12	14
Cluster-threshold	0.8	0.8	2.0	2.0	2.0	0.8
Recursion-threshold	0.35	0.85	0.75	0.85	0.85	0.35
SAT-connectivity	25	25	3	25	25	25
Side-connectivity	0.8	0.8	0.8	0.8	0.8	2.5
Area (% of original)	76%	40%	160%	135%	80%	88%
Wire-length (% of orig.)	37%	28%	145%	175%	104%	70%
Aspect ratio	1.005	1.2	2.1	1.1	1.07	1.28

Figure 6.43: Overall performance summary

detailed router within the recursive-build algorithm.

We now consider each of the major weight factors in turn. The table shows that, for the cluster-threshold factor, the examples fall into two groups: *EX1*, *EX2* and *EX6* in one, with a cluster-threshold of 0.8; and the others with a cluster-threshold of 2.0. Looking at the dimensions for *EX1*, *EX2* and *EX6*, we see that they are more uniform, with less variation than the others. Hence, by allowing larger clusters (which is the effect of lowering the cluster-threshold), more modules are made available to the SAT selection process, which improves the chances of a good placement. The other group of layouts, *EX3*, *EX4* and *EX5*, have a wider spread in their dimensions, and hence putting only the more tightly-coupled modules together in a cluster prevents poorly matched modules from being built together at an early stage. A guide for selecting reasonable cluster-threshold weights might therefore be the spread of module dimensions, with more uniform dimensions indicating that a lower cluster-threshold may be used. In any case, the range of successful cluster-thresholds is small enough that a limited search could be undertaken in that range for a good layout.

Recursion-threshold values split the layouts into three groups corresponding to the values 0.35, 0.75 and 0.85; We consider the latter two groups to be actually in the same group, since the recursion-threshold values are close enough. *EX3* is the lone member of group 2; its minimum and maximum dimensions are also the farthest apart, which might have a bearing on this factor. However, we should also note that many layouts were disqualified on the basis of aspect ratio alone, since an aspect ratio of above 3 is unacceptable enough to force even low-area layouts to be discarded (see Figures 6.10 — 6.16). The main reason for the choice of final recursion-thresholds being the aspect ratio, we therefore consult the range of dimensions of the modules again. Modules *EX3*, *EX4* and *EX5* contain the widest spread of dimensions, while modules *EX2*, *EX4* and *EX5* have the lowest count of nets per module (all around or below 2, compared to nearly 5 for the rest). Widely differing dimensions would be one good reason to require a high recursion-threshold leading to long sequences of recursive-building to build up the smaller modules until they are large enough. Alternatively, if connectivity is generally low, then again a high degree of recursion may be in order, since the channels will not be very dense and will not need much routing area.

For *EX1* and *EX6*, with high connectivity, it is quite likely that repeated recursive building (which takes place without any re-clustering in between) causes the following problem with orientation. Some layouts do not use all four sides of modules uniformly in terms of terminal locations; it is often the case that, for instance, the top and bottom sides of modules have no terminals situated on them. However, if all sides of the modules *do* have terminals located on them, then no matter which orientation is chosen, large channels have to be built. In this situation, therefore, the best policy is to stop recursive-building from building up too many modules together outside the control of the clustering phase. We note that for these examples, a low recursion-threshold is combined with a low cluster-threshold, allowing large clusters and not too much recursion.

The final two factors have very little variation between layouts. SAT-connectivity is very successful at a value of 25, with only *EX3* requiring a deviation to 3. Since *EX3* has the largest number of nets with approximately the same number of modules, it would need a lower emphasis on connectivity than the other modules (remember that the degree-of-fit function returns values that are constrained to be small integers, whereas the converted SAT-connectivity value depends on the number of terminals on a module, which is as high as 34 out of a total of 89 nets for this example). Finally, side-connectivity is uniformly 0.8 for the first five examples, and 2.5 for the last one: the probable cause is its combination of large degree of connectivity and small range of dimensions. In this situation, the orientation function is not able to obtain a wide-enough separation between the different edge-dimensions available, and hence needs to emphasize the connectivity criterion to help make better orientation decisions.

In chapters 4 and 5, example layout *EX1* was repeatedly used as a real-world

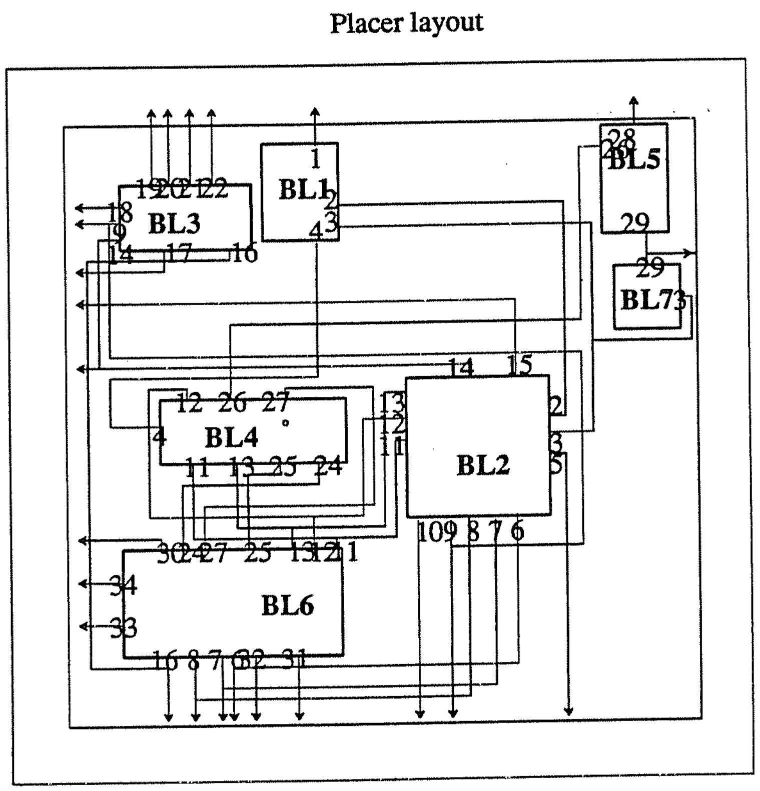


Figure 6.44: Final Placer layout for *EX1* (outer box = original size).

example to illustrate the working of our algorithm. Figure 6.44 displays the final result for *EX1* obtained by the Placer, displayed at the same scale as the original layout (Fig. 4.2). The outer box bounding this layout shows the size of the original placement, while the inner box is the bounding box specified by the Placer for its final layout. The one advantage that the original layout has over the Placer layout is that its external nets have been arranged so that connections to input/output pads can easily be done. On the other hand, much can be said in favor of the Placer's layout. The most striking difference is the drastic reduction in the amount of wiring required. The Placer appears to have arranged the modules so as to cut down on the amount of long wiring needed, which is the effect that was targeted by the clustering algorithm. Secondly, it can be clearly seen that all the channels provided by the Placer are very liberally estimated; there is no channel that is carrying its capacity of nets. For instance, the channel between modules BL3 and BL4 has enough space to route 10 tracks, whereas only 6 are actually needed. Finally, the placement obtained is itself far from satisfactory; it is clear that much more improvement can be obtained by moving module BL2 down, making room for modules BL5 and BL7 in the upper right corner to be compacted into the vacant area to their left. It is quite clear that a compaction stage would make a significant improvement in the final layout.

In summary, we have carried out exhaustive tests with the Placer, and have obtained outstanding results on 4 of the 6 sample layouts available. The Placer was able to produce layouts which were as low as 40% of the area of the original layout, with only two of the examples not doing as well as the original layouts. Wire-length and aspect ratio performance was also very good as compared to the original layouts. The set of weights to be applied has been brought down to a very small subset, and reasonable guidelines for using the weights have been evolved. Moreover, the Placer being greedy and fast, a limited search of the available set of weight-factor combinations is also possible with very little computing time consumed in comparison to the computation times of the conventional global placement and routing approaches.

Chapter 7

SUMMARY AND CONCLUSIONS

Considering the problem of placing and routing a set of modules on a VLSI chip, a greedy approach was suggested as a fast and approximate method of obtaining results that might be comparable to layouts obtained by much more time-bound global approaches. The problem, defined as the placement of a set of rectangular modules on a chip and the routing of wires between terminals on the boundaries of the modules, had been shown to be NP-hard. Therefore, attempts to develop algorithms for obtaining optimum placements have been abandoned in favor of obtaining acceptable placements.

Currently successful conventional approaches to the placement and routing problem have fallen into three broad classes of algorithms: top-down partitioning, bottom-up cluster-growth and combined approaches. Top-down partitioning uses the idea of min-cut to partition the module set into subsets which are likely to minimize the wiring to be done between them, while cluster-growth builds a placement by putting highly-connected modules near each other. Min-cut has the drawback of having to solve the NP-complete sub-problem of finding an even division of the module set that also minimizes the connections between the partitions before proceeding with the placement. The cluster-growth approach, on the other hand, often leads to congested channels near the center of the placement area because it is based mainly on local connectivity information without considering global concerns. Combined approaches have had some success, but nearly all of them suffer from very high computational times, since they are all based on some form of backtracking in order to obtain successful layouts. Concerning the problem of global wire routing, conventional approaches tend to treat this as a separate phase, to be applied after completion of the placement phase.

Greedy placement and incremental global routing is an attempt to overcome the backtracking nature of conventional algorithms. The idea of global routing as a separate phase is discarded in favor of combining placement and routing into a single process, in order to provide the router with the information available during the placement stage. In this framework, a clustering approach is combined with recursive building of the layout. Clustering is based on a novel scheme that identifies *CORE* modules as those that are considered to be most important by the largest number of *satellite* modules. Both selection and orientation of modules is influenced by a combination of connectivity and dimensional considerations. The layout is built step-by-step, adding on modules to clusters one at a time. Global routing is performed incrementally, completing just enough of the wire-routing at each step to be able to completely specify a partially laid out set of modules as a *supermodule*, with all external terminals routed from their internal source-modules to the periphery of the supermodule. The decision-making process for such things as the membership of the clusters, the point at which to recursively build up a pair of modules into supermodules, the choice of modules to build up and their orientations, are all driven by a set of weights assigned to each decision function.

The Placer was extensively tested over a very large parameter space. Six test layouts were obtained from examples in journal publications, and these layouts were used as inputs to the Placer. The tests revealed that a very small subset of weight values produced the best layouts for all the sample problems. The Placer was also successful in producing layouts that were better than the original layouts in 4 of the 6 cases. The tests for the individual decision criteria were able to establish that each of the criteria was worth using, by showing that layouts produced by ignoring the factors failed to do as well as those that took all the factors into consideration. The algorithm proved to be very fast, placing and routing the samples in real-time periods of between ½ a minute for a 7-module layout, and 3½ minutes for a 19-module layout.

The tests also revealed some areas of possible future research. Considering that most current-day installations are configured as local-area networks containing many work-stations, the greedy approach fits in very well with such an environment. Given the set of weights to be applied, each layout task is totally independent of the others. This therefore provides us with a natural division of the overall series of Placer runs into convenient units for parallelization and independent execution on separate processors. Minimal additional coding would be sufficient in order to convert a series of Placer runs into a parallel-Placer run. Since the Placer is also very fast in producing its results, such a configuration would be able to yield real-time placements much faster than existing systems, with comparable results. Moreover, since the testing of the Placer involved a series of runs to determine the best layouts and the corresponding weights, such a search could be combined with the Placer in the manner of a simulated annealing exercise. The Placer itself could be run at a set of weights, using the hill-climbing techniques of simulated annealing to locate good layouts.

Turning to the algorithms of the Placer, we note that one of the characteristics of the layouts was a tendency for aspect ratios to vary in an extreme manner. The reason for this is that the Placer does not directly control aspect ratio, leaving it to the recursive-building and orientation algorithms to control in their indirect manner. It would seem that this approach is not sufficiently successful, since many otherwise good layouts had to be discarded because of poor aspect ratios. A separate control for aspect ratio, however, is likely to fail, since it is possible for supermodules to have very poor aspect ratios until near the very end of the placement process; the last few module-building steps might successfully put together some high aspect-ratio modules to produce a final layout with a very good aspect ratio. Hence, a control for aspect ratio must be based not only on the aspect ratios of individual supermodules, but also on the number of modules (and clusters) currently present in the module set. Such a control should be constructed to encourage better aspect ratios close to the end of the layout process, remaining dormant during earlier stages.

A second area that we feel needs further investigation, is the Recursive-build algorithm. This algorithm allowed only SAT modules to be built, and not the CORE module. The reason for this was that some restraint needed to be placed on the degree of recursive building. Allowing both CORE and SAT modules to build themselves up whenever either one was smaller than the other might lead to unconstrained recursive building beginning with a single recursive call, bypassing all re-clustering phases. While that is a valid problem, the restraint, in its turn, leads to wasted area because of built-up SAT modules that are much larger than their unchanged CORE modules. Some limited degree of building-up of the CORE module might provide an improvement. In effect, this means that in certain circumstances, a non-recursive, strictly limited building of CORE might be sanctioned; hence an additional non-recursive module-building function needs to be implemented.

Having had the experience of the tests that established the performance and the weight-assignment guidelines for the Placer, we feel that the next step would be possible if similar tests could be carried out on a very large number of sample layouts. It would then become possible to issue very detailed guidelines for the use of the weights for different kinds of layout problems. Moreover, it might be possible to obtain some amount of correlation between the architecture of the chip being placed and the set of weight-ranges to use. Such a correlation might be based on human experience, as well as on the characteristics of the modules constituting typical examples of such architectures. It might then be possible for the Placer to generate a set of layouts given additional information about the class of architecture that the problem belonged to.

Finally, in terms of converting our experimental Placer into a production-quality system, we envisage two additional projects that would complete the job. First, a detailed channel router is necessary to convert the Placer into a production-quality system. Such a router would need to be supported by input of a more detailed description of the module set to be placed. This might also bring down the area of the layouts by replacing the estimated channel areas by their actual areas. Second, a placement improvement and compaction stage could be added to the Placer in order to squeeze some additional improvement from the final layouts.

BIBLIOGRAPHY

[Aho83a] Aho, A. V., J. E. Hopcroft, and J. D. Ullman, *Data Structures and Algorithms,* Addison-Wesley, Reading, MA (1983).

[Anto85a] Antognetti, P., A. De Gloria, G. Arato, and O. Gaiotto, ''ARIANNA: A Floor-Planning Tool,'' *Proc. 11th European Solid State Circuits Conf.*, pp. 193-200 (1985).

[Blan84a] Blanks, J. P., ''Initial Placement of Gate Arrays using Least-Squares Methods,'' *Proc. 21st Design Automation Conference,* pp. 670-671 (1984).

[Cies87a] Ciesielski, M. J., ''Digraph Relaxation for 2-Dimensional Placement of IC Blocks,'' *IEEE Transactions on Computer-Aided Design* **CAD-6**(1) pp. 55-66 (January 1987).

[Clow84a] Clow, G. W., ''A global routing algorithm for general cells,'' *Proc. 21st Design Automation Conference,* pp. 45-51 (1984).

[Deut76a] Deutsch, D. N., ''A Dogleg Channel Router,'' *Proc. 17th Design Automation Conference,* pp. 425-433 (1976).

[Dona80a] Donath, W. E., ''Complexity Theory and Design Automation,'' *Proc. 17th Design Automation Conference,* pp. 412-419 (1980).

[Gare79a] Garey, M. R. and D. S. Johnson, *Computers and Intractability,* Freeman & Co. (1979).

[Goto79a] Goto, S., ''Layout Problem,'' *Proc. 16th Design Automation Conference,* pp. 11-17 (1979).

[Hana72a] Hanan, M. and J. M. Kurtzberg, ''A Review of the Placement and Quadratic Assignment Problems,'' *SIAM Review* **14**(2) pp. 324-342 (April 1972).

[Hana72b] Hanan, M. and J. M. Kurtzberg, ''Placement Techniques,'' pp. 213-282 in *Design Automation of Digital Systems, Volume One: Theory and Techniques*, ed. M. A. Breuer,Prentice-Hall, Inc. (1972).

[Hart86a] Hartoog, M. R., ''Analysis of Placement Procedures for VLSI Standard Cell Layout,'' *Proc. 23rd Design Automation Conference*, pp. 314-319 (1986).

[Hill80a] Hillier, F. S. and G. J. Lieberman, *Introduction to Operations Research*, Holden-Day, Inc., San Fransisco (1980).

[Kurt65a] Kurtzberg, J. M., ''Algorithms for Backplane Formation,'' in *Microelectronis in Large Systems*, Spartan Books, Washington, D.C. (1965).

[Laut79a] Lauther, U., "A Min-Cut Placement Algorithm for General Cell Assemblies Based on Graph Representation,'' *Proc. 16th Design Automation Conference*, pp. 1-10 (1979).

[Lawl66a] Lawler, E. L. and D. E. Wood, ''Branch-and-bound Methods: a Survey,'' *Operations Research* **14**(4) pp. 699-719 (July-August 1966).

[Lee61a] Lee, C., ''An algorithm for path connections and its applications,'' *IRE Trans. on Electronic Computers*, pp. 346-365 (Sep. 1961).

[Leis80a] Leiserson, C. E., ''Area-Efficient Graph Layouts (for VLSI),'' *Proc. 13th IEEE Symposium on Foundations of Computer Science*, pp. 270-281 (1980).

[Loos79a] Loosemore, K. J., ''Automated layout of integrated circuits,'' *Proc. 1979 IEEE Symp. Circuits and Systems*, pp. 665-668 (1979).

[Marg87a] Margarino, A., A. Romano, A. De Gloria, F. Curatelli, and P. Antognetti, ''A Tile-Expansion Router,'' *IEEE Transactions on Computer-Aided Design* **CAD-6**(4) pp. 507-517 (July 1987).

[Mead80a] Mead, C. A. and L. A. Conway, *Introduction to VLSI Systems*, Addison-Wesley (1980).

[Moha87a] Mohan, M. K., ''Using the Greedy Placer and Incremental Global Router,'' *Technical Report, in preparation*, (August 1987).

[Moor59a] Moore, E. F., ''Shortest path through a maze,'' *Annals of the Computation Lab. of Harvard University* **30** pp. 285-292 Harvard University Press, (1959).

[Mura80a] Murai, S., M. Kakinuma, M. Imai, and H. Tsuji, ''The Effects of the Initial Placement Techniques on the Final Placement Results,'' *Proc. of the IEEE Conference on Circuits and Computers*, pp. 80-82 (1980).

[Nils71a] Nilsson, N. J., *Problem solving methods in Artificial Intelligence*, McGraw-Hill (1971).

[Pint83a] Pinter, R. Y., ''River Routing: Methodology and Analysis,'' pp. 141-164 in *Proc. 3rd CALTECH Conference on VLSI*, Computer Science Press (1983).

[Prea78a] Preas, B. T. and C. W. Gwyn, ''Methods for Hierarchical Automatic Layout of Custom LSI Circuit Masks,'' *Proc. 15th Design Automation Conference*, pp. 206-212 (1978).

[Prea86a] Preas, B. T. and P. G. Karger, ''Automatic Placement: a Review of Current Techniques,'' *Proc. 23rd Design Automation Conference*, pp. 622-629 (1986).

[Quin75a] Quinn, N. R., "The Placement Problem as viewed from the Physics of Classical Mechanics,'' *Proc. 12th Design Automation Conference*, pp. 173-178 (1975).

[Reed85a] Reed, J., A. Sangiovanni-Vincentelli, and A. Santamauro, ''A New Symbolic Channel Router: YACR2,'' *IEEE Transactions on Computer-Aided Design* 4(3) p. 208 (July 1985).

[Rich84a] Richard, B. D., ''A Standard Cell Initial Placement Strategy,'' *Proc. 21st Design Automation Conference*, pp. 392-398 (1984).

[Rive82b] Rivest, R. L. and C. M. Fiduccia, "A "Greedy" Channel Router,'' *Proc. 19th Design Automation Conference*, pp. 418-424 (1982).

[Rive82a] Rivest, R. L., ''The PI (Placement and Interconnect) Project,'' *Proc. 19th Design Automation Conference*, (1982).

[Roth83a] Rothermel, H. and D. A. Mlynski, ''Automatic Variable-Width Routing for VLSI,'' *IEEE Transactions on Computer-Aided Design* **CAD-2**(4) pp. 271-284 (October 1983).

[Sahn80a] Sahni, S. and A. Bhatt, *Proc. 17th Design Automation Conference*, pp. 402-411 (1980).

[Sech86a] Sechen, C. and A. Sangiovanni-Vincentelli, ''TimberWolf3.2: A New Standard Cell Placement and Global Routing Package,'' *Proc. 23rd Design Automation Conference*, pp. 432-439 (1986).

[Souk81a] Soukup, J., "Circuit Layout," *Proc. of the IEEE* **69**(10) pp. 1281-1304 (1981).

[Szym85a] Szymanski, T. G., "Dogleg Channel Routing is NP-complete," *IEEE Transactions on Computer-Aided Design* **CAD-4**(1) pp. 31-40 (January 1985).

[Ullm84a] Ullman, J. D., *Computational Aspects of VLSI,* Computer Science Press (1984).

[Wipf82a] Wipfler, G. J., M. Wiesel, and D. A. Mlynski, "A Combined Force and Cut Algorithm for Hierarchical VLSI Layout," *Proc. 19th. Design Automation Conference*, pp. 671-677 (1982).

[Yosh82a] Yoshimura, T. and E. S. Kuh, "Efficient Algorithms for Channel Routing," *IEEE Transactions on Computer-Aided Design* **1**(1) pp. 25-35 (1982).