VLSI Design
1994, Vol. 2, No. 2, pp. 117–129
Reprints available directly from the publisher
Photocopying permitted by license only

An Effective Solution to the Linear Placement Problem

YOUSSEF SAAB and CHENG-HUA CHEN
Computer Science Department, University of Missouri-Columbia, Mathematical Sciences Building, Columbia, MO 65211

We present an effective solution to the linear placement problem, which has several applications in the physical design of integrated circuits. Our approach belongs to the class of iterative improvement heuristics. The important difference of this new technique from the previous ones is in its moves, and in the order of application of these moves. A phase of the algorithm begins with simple moves and gradually shifts towards more complex moves. Phases are repeated as long as further improvement is possible. Our experimental results show that nearly optimal solutions can be achieved. For a number of examples collected from the literature, our algorithm generated optimal solutions.

Key Words: *Linear placement; Linear ordering; Board permutation; Back-board ordering; String placement; 1-dimensional gate assignment*

1 INTRODUCTION

The linear placement problem (LPP) occurs in different forms in the physical design automation of integrated circuits. Various names were used to describe LPP: linear ordering [1], board permutation [2], back-board ordering [3, 4, 5], string placement [6], and 1-dimensional gate assignment [7]. According to Yamada et al. [7], LPP is regarded by some researchers as a fundamental and significant problem in the layout of VLSI circuits.

A linear placement is the arrangement of a number of interconnected circuit elements in a row so that a certain objective is met. Various applications of LPP have been reported in the literature. In [1], Kang uses LPP in a constructive initial placement for standard cells. The cells are initially arranged in one long row, which is then folded into a 2-dimensional placement. Another approach to the placement of standard cells is to assign the cells to rows in a first step. In a second step, the order of the cells in their respective rows is determined by solving a linear placement problem in each of the rows. The latter approach has been used by Cho and Kyung [8]. In gate array design style, Chowdhury [9] mentions that a good placement may be achieved by initially assigning the gates to rows and columns. Then, the placement is improved by permuting the columns to minimize the horizontal wire length.

The linear placement problem is indeed a special case of the more general 2-dimensional placement problem. Therefore, in principle, techniques for the latter problem can be used to solve LPP. However, there is rarely any mention in the literature of such approaches. To the contrary, techniques for LPP have been used to solve the more difficult 2-dimensional placement problem [1, 8]. Because LPP is an NP-Hard problem [14], most algorithms in the literature are heuristics in nature [1–7, 9–13]. In [4], Goto et al. provide a branch-and-bound algorithm which generates a solution whose cost is no more than $(1 + \varepsilon)$ times the optimal cost. However, the computational effort of Goto's algorithm increases exponentially as ε decreases. In [1], Kang presents a constructive algorithm, which places the circuit elements one at a time. The most lightly connected circuit element is placed in the first position. Subsequently, based on a selection rule, an unplaced element is chosen and is placed in the next vacant position. This process is repeated until all the circuit elements have been placed. In [11], Schuler and Ulrich give an algorithm, which divides the circuits elements into several clusters. Subsequently, the circuit elements are arranged in a row, such that elements

of the same cluster are kept close to each other. In [12], Cheng obtains a linear placement method using the max-flow min-cut theory [15]. Cheng's algorithm requires for its input a circuit which has the topology of a graph. For parallel graphs, Cheng proves that his algorithm finds an optimal solution in polynomial time. Previously, Adolphson and Hu [13] used network flow theory to obtain an algorithm which finds an optimal solution in $O(nlog\ (n))$ operations if the input graph is a rooted tree. In [9], Chowdhury uses nonlinear programming techniques to solve LPP. His algorithm starts at a point deep inside the infeasible region and proceeds gradually towards the feasible region along the trajectories in which the cost tend to be minimized. In [7], Yamada et al. offer a hierarchical algorithm for LPP based on contraction of nets. In this approach, the original problem is partitioned into a number of levels in a bottom-up contraction of multiterminal nets, such that nets with fewer terminals are given first priority. After that, the elements of the circuits are assigned to positions in the linear order in a top-down manner.

The algorithm described in this paper is iterative and can be thought of as an incremental clustering approach. The early stages of each phase of this algorithm form and rearrange small clusters in the circuit, while later stages rearrange larger clusters that were formed during the previous stages. In the algorithm, any set of consecutive nodes in the linear placement can be a cluster, i.e., no other criterion is used in the definition of a cluster. The hypothesis is that as the linear placement improves, larger clusters appear as a sequence of consecutive nodes in the placement. Further improvement of the linear placement can then be achieved by the rearrangement of these large clusters. Phases of the algorithm are repeated as long as further improvement can be achieved. One key feature of this algorithm is that nodes of large clusters are allowed to break away from each other in later stages. This feature should be compared to the approach used by Schuler and Ulrich [11]. The algorithm of Schuler and Ulrich consists of two steps: clustering and linear placement. In the clustering step, larger clusters are recursively formed from previous smaller clusters. This step results in the so-called clustering tree. Each vertex in this tree corresponds to a cluster of nodes of the circuit. The cluster of an internal vertex of the tree is the union of the clusters of its left and right children. After the formation of the clustering tree, a linear placement can be obtained by a left to right reading of the nodes corresponding to the leaves of the tree. However, this placement may not be good. Therefore, Schuler and Ulrich consider improving this placement by rotating subtrees of the clustering tree around their roots. The rotation of subtrees can be done in two ways: top-down or bottom-up. In the top-down approach, larger clusters are placed first, and then the placement is improved by rearranging nodes within the same cluster. In the bottom-up approach, the sub-clusters of larger clusters are arranged before the arrangement of the larger clusters. Note that in both the top-down and bottom-up approaches, nodes of the same cluster must stay together. This is a disadvantage, since the number of distinct placement that can be reached from a given clustering tree is $2^{(n-2)}$ which is a small fraction of all $n!/2$ different placements. In comparison with Schuler and Ulrich algorithm, each phase of our algorithm can be considered as a bottom-up approach in which the clustering and the linear placement steps are inter-mixed. The advantage of our approach over the approach of Schuler and Ulrich is that nodes of large clusters are allowed to break away from each other. This removes the limitation on the number of distinct placements that can be explored. In fact, later in this paper, it will become clear that any two linear placements are reachable from each other through the moves used in our algorithm.

2 PRELIMINARIES

An electrical circuit consists of a set V of n elements, and a multiset E, which consists of subsets of V. In the literature, the elements of V are given different names by different authors. Modules, nodes, gates, and cells are just few of the names that are used to describe the elements of V. Each element of E is a subset of two or more elements of V, and it represents a physical electrical connection of its constituent elements. The elements of E are formally called hyperedges, but are commonly called nets. In this paper, we refer to the elements of V as nodes, and to the elements of E as nets.

To simplify the notation, the set V is taken to be the set of the first n positive integers, i.e., the integer $1 \leq i \leq n$ is the name of the i-th node in V. It should be clear from the context whether an integer i is used as the name of a node in V, or as a numerical value. The j-th net in E, is denoted by N_j, $1 \leq j \leq m$, where $m = |E|$ is the number of nets. With this notation, a permutation is a one-to-one mapping π: $V \rightarrow V$. If $\pi(i) = j$, then we say that node j is in position i with respect to the permutation π. Given a permutation π of V, define:

(1) $l(\pi, j)$ $(h(\pi, j))$ to be the minimum (maximum) integer in the set $\{i: \pi(i) \in N_j\}$, i.e., $l(\pi, j)$

$(h(\pi, j))$ is the leftmost (rightmost) position of a node in net N_j.

(2) $cut(\pi, i)$ to be the number of nets in the set $\{N_j: l(\pi, j) \leq i < h(\pi, j)\}$.

(3) $D(\pi)$ to be the maximum element of the set $[cut(\pi, i): 1 \leq i < n\}$. The parameter $D(\pi)$ is the density of the permutation π.

(4) $L(\pi)$ to be the sum of the element of the set $\{cut(\pi, i): 1 \leq i < n\}$. The parameter $L(\pi)$ is the length of the permutation π. Note that $L(\pi) = \Sigma_{j=1}^{m} (h(\pi, j) - l(\pi, j))$.

Consider the circuit consisting of 8 nodes as shown in Figure 1. The nodes of this circuit have been arranged in a row according to a permutation π. In this permutation π, the position of node 6 is 3 ($\pi(3) = 6$). The number of nets crossing the dashed vertical line is $cut(\pi, 4) = 3$. The leftmost (rightmost) position of a node in N_1 is $l(\pi, 1) = 3$ ($h(\pi, 1) = 8$). The density of π is $D(\pi) = 3$, and the length of π is $L(\pi) = 16$.

Given a circuit (V, E), a linear placement problem (LPP) seeks a permutation π, which minimizes a certain cost function. One possible cost function is the density $D(\pi)$ of the permutation. The number of tracks required to route the nets is at least equal to $D(\pi)$. Therefore, the minimization of $D(\pi)$ leads to a smaller number of routing tracks. Another cost function is the length $L(\pi)$ of the permutation. The total wire length required to route the nets is at least equal to $L(\pi)$. Consequently, the minimization of $L(\pi)$ leads to a shorter wire length. A third cost function is $\Sigma_{j=1}^{m} (h(\pi, j) - l(\pi, j))^2$. This cost function is similar to the second one. However, it places more weights on the minimization of longer wires. According to Yamada et al. [7], better layout can be achieved if shorter wires are given precedence. Taking this point of view, the third cost function may not be a good choice. The advantage of the third cost function is the fact that it is mathematically well-behaved, and is therefore suitable for analytical methods [9]. In this paper, the length $L(\pi)$ of the permutation is used as a cost function. However, our algorithm is not dependent on the particular cost function used, and any other cost function may be used instead. In [7], it is shown that the layout area is linearly dependent on the length of the permutation. Therefore, the minimization of $L(\pi)$ leads to smaller overall area.

3 THE ALGORITHM

An iterative improvement algorithm starts with some initial solution, which is then improved by a local transformation called a *move*. The new solution is then used as the initial one, and the process is iterated until no further improvement is possible.

Several iterative improvement algorithm for LPP have been reported in the literature [2, 6, 10]. In [16], Goto introduces the notion of λ-optimality. For LPP, one may call a λ-move any move that involves

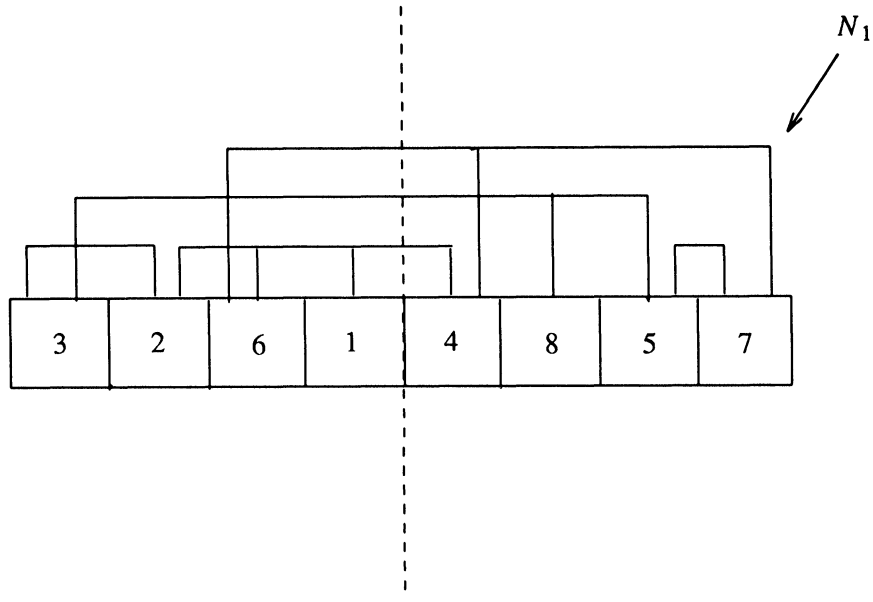

FIGURE 1 A circuit with 8 nodes.

the rearrangement of λ nodes of the circuit. If there are no λ-moves that can reduce the cost, the solution is called λ-optimal. Clearly, an n-optimal solution is optimal. For a given $\lambda > 1$, there are $\binom{n}{\lambda}\lambda!$ possible λ-moves. Consequently, if it is desired to guarantee λ-optimality, the computational effort of an iterative improvement algorithm increases exponentially with λ. Hence, the most commonly used value for λ is 2 [9]. In this case, the iterative improvement algorithm is better known as a two-exchange strategy.

Two-exchange algorithms for LPP are easily trapped in local minima. Consider a circuit of n nodes and $n - 1$ nets, such that net N_i connects the nodes i and $i - 1$. Let us call such a circuit a *chain*. Clearly, up to a reversal of order, the only optimal permutation for a chain is the permutation π given by $\pi(i) = i$, $1 \leq i \leq n$. For the chain of n nodes initially placed as in Figure 2, and for large enough n, no two nodes can be exchanged such that the cost is strictly reduced. The cost of the permutation in Figure 2 is equal to $n/2 + 2(n/2 - 1) = 3n/2 - 2$, while the optimal cost is $n - 1$. This illustrates that even for a simple chain circuit, a two-exchange algorithm may be trapped in a local minimum which is far away from the globally optimal solution. Moreover, even if moves which increase the cost are occasionally accepted as in simulated annealing [17] or stochastic evolution [10, 18], a two-exchange algorithm may spend a long computation time before reaching the optimal solution.

The exchange of the positions of two nodes is an example of a *simple* move that changes the permutation only slightly. Formally, if one defines a distance measure on the space of all permutations then a new permutation generated by a simple move is only a short distance away from the previous one. Suppose that $\pi_1, \pi_2, \ldots, \pi_k$ is a sequence of permutations in the order of increasing distance from a reference permutation π_0. Suppose also that this sequence is such that π_i is obtained from π_{i-1} by a simple move. Let $c_1, c_2, \ldots, c_k$ be the sequence of costs corresponding to the sequence of permutation. We can now plot a sample cost-versus-permutation curve using the sequence of coordinates $(d_1, c_1), (d_2, c_2), \ldots, (d_k, c_k)$, where d_i is the distance of π_i from the reference permutation π_0. If the cost-versus-permutation curve is as in Figure 3, then a simple move is a downhill walk along this curve from the starting permutation. Consequently, given the shape of the curve, it is clear that a simple move may not go a long way before it is stuck at a local minimum. This discussion suggests that if a near-optimal solution is sought, then an iterative improvement algorithm should use moves other than simple ones. The kind of moves that are suggested here are illustrated in Figure 4 in which the arrow represents a possible move. These moves are not simply a walk along the cost curve. Rather, they constitute jumps from one point on the curve to another. Such moves can be called *compound* moves, and they involve the rearrangement of a large number of nodes in the circuit. Here, a conflict is reached. In order to find near-optimal solutions, one have to consider compound moves in which a large number of nodes are rearranged. However, if all possible moves that involve the rearrangement of a large number of nodes are considered, the computational effort of the algorithm grows exponentially. To remedy this situation, the algorithm should only consider few compound moves that are meant to produce large improvements in return for the computational expense of using them.

The method used by our algorithm can best be called a limited exhaustive search strategy (LESS), since effort is made to choose small polynomial number of moves out of exponentially many possible ones. The algorithm itself is also called LESS.

The idea which lead to the development of LESS is now explained. Given a permutation π of the nodes of the circuit, any set of nodes occupying consecutive positions is called a block. Clearly, any two integers

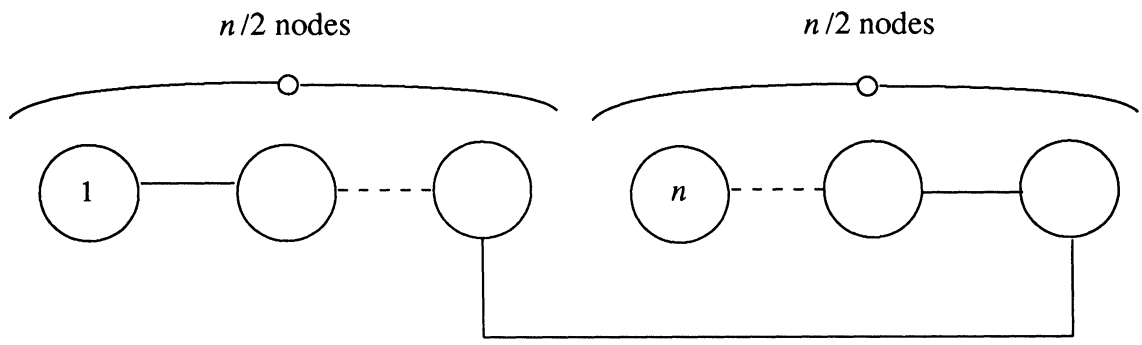

FIGURE 2 A possible arrangement for a chain of n nodes.

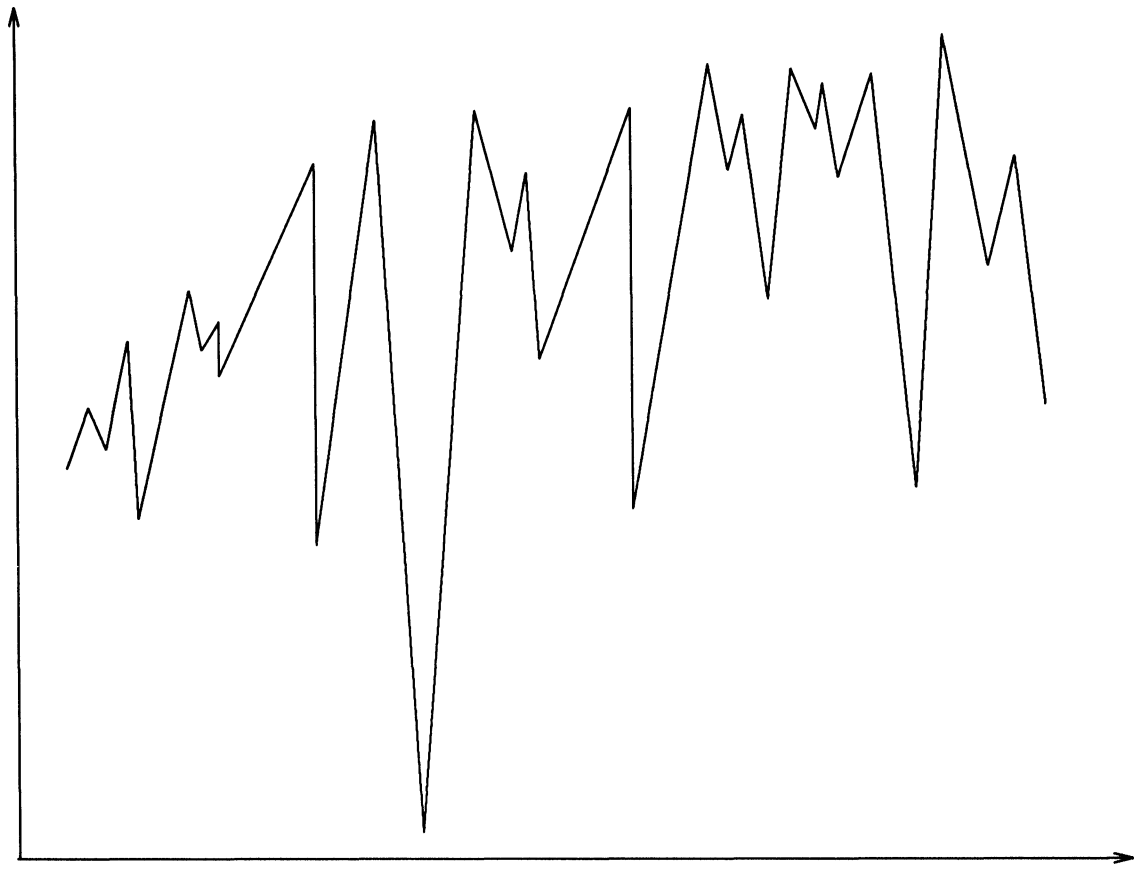

FIGURE 3 A hypothetical cost-versus-permutation curve.

i and j such that $1 \leq i \leq j \leq n$ determine the block $B_{ij} = \{\pi(k): i \leq k \leq j\}$. Hence in a permutation, there are $(n^2 + n)/2$ blocks. Suppose that an iterative improvement algorithm for LPP has been used to determine a permutation π. If at all any improvement has been achieved, it would be that the permutation π contains a number of blocks which should also appear in an optimal permutation. If π is not optimal, it is likely that these good blocks are either placed in the wrong positions and/or they are placed in the wrong order. Our problem now is that it is not known which set of consecutive nodes in π is a good block. One strategy for improving π is to consider every block of nodes at a time and try to place it in a position which yields the maximum reduction of cost. The length of a block is defined to be the number of nodes in that block. Clearly, the longer the good blocks are, the better is the permutation π. The strategy in LESS is to consider the initial permutation as one which is far away from an optimal solution. Therefore, the only possible good blocks at this stage are those of length 1, i.e., every node constitute a block. Hence initially, LESS considers every node and tries to place it in the position which yields the maximum improvement. After all nodes have been considered, the permutation has been improved considerably that it may contain many good blocks of length 2. Consequently, blocks of length 2 are considered next. In general, at the i-th stage of LESS, the blocks that are considered have length i. This process continues until all blocks have been considered. This completes a phase of LESS. Phases are repeated until no further improvement is possible.

There are three types of moves used by LESS: FLIP, TRANSFER, and TRANSFER_FLIP. The move FLIP(i, j, π) takes the block B_{ij} and reverses the order of nodes in it. More precisely, this move transforms the current permutation π into a new permutation π' such that $\pi'(l) = \pi(i + j - l)$ if $i \leq l \leq j$ and $\pi'(l) = \pi(l)$ otherwise. The move TRANSFER(i, j, k, π) takes the block B_{ij} and places it after position k in the permutation. If $k < i - 1$, then the nodes in positions $k + 1$ up to $i - 1$ are shifted to the right to make room for the nodes in B_{ij}. If $k > j$, then the nodes in positions $j + 1$ up to k are shifted to the left to make room for the nodes in B_{ij}. If $k =$

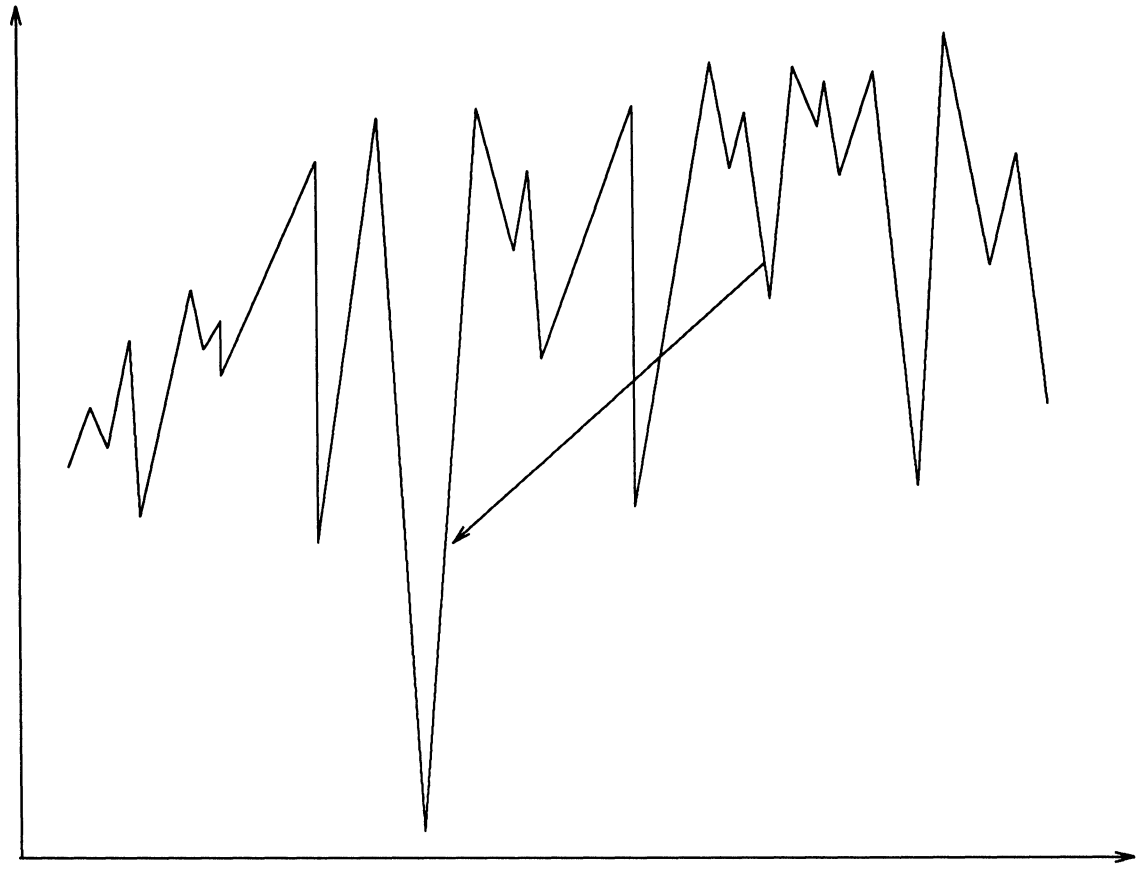

FIGURE 4 An illustration of a compound move.

$i - 1$, then TRANSFER(i, j, k, π) leaves the permutation unchanged. Note that k can be equal to 0. If $k = 0$, then the block B_{ij} is placed in front of the remaining nodes in the permutation. Without loss of generality, assume $k > j$. The new permutation π' created by TRANSFER(i, j, k, π) is such that $\pi'(l) = \pi(l)$ if $l < i$ or $l > k$, $\pi'(l) = \pi(l + j - i + 1)$ if $i \le l < i + k - j$, and $\pi'(l) = \pi(l - k + j)$ if $i + k - j \le l \le k$. The move TRANSFER_FLIP(i, j, k, π) is similar to TRANSFER(i, j, k, π) except for the reversal of the order of the nodes of B_{ij}. The move TRANSFER_FLIP(i, j, k, π) takes the block B_{ij} and places it in reversed order after position k in the permutation. Without loss of generality, assume $k > j$. The new permutation π' created by TRANSFER_FLIP(i, j, k, π) is such that $\pi'(l) = \pi(l)$ if $l < i$ or $l > k$, $\pi'(l) = \pi(l + j - i + 1)$ if $i \le l < i + k - j$, and $\pi'(l) = \pi(i + k - l)$ if $i + k - j \le l \le k$. It should be understood that after any of the above moves the permutation π is updated to reflect the change. Now, the algorithm LESS can be described as follows:

Step 1. Generate an initial permutation π.
Step 2. Set *length* = 1 and set *improve* = *false*.
Step 3. Set $i = 1$.
Step 4. Set $j = i + length - 1$.
Step 5. Find position $k \notin [i - 1, j]$ such that the move TRANSFER(i, j, k, π) yields a maximum reduction *gain* in cost.
Step 6. If *gain* > 0 then perform the move TRANSFER(i, j, k, π) and set *improve* = *true*.
Step 7. If $j < n$ then set $i = i + 1$ and go to Step 4.
Step 8. If *length* = 1 then set *length* = *length* + 1 and go to Step 3 (The moves FLIP and TRANSFER_FLIP do not make sense unless *length* > 1).
Step 9. Set $i = 1$.
Step 10. Set $j = i + length - 1$.
Step 11. Find position $k \notin [i - 1, j]$ such that the move TRANSFER_FLIP(i, j, k, π) yields a maximum reduction *gain* in cost.
Step 12. If *gain* > 0 then perform the move TRANSFER_FLIP(i, j, k, π) and set *improve* = *true*.
Step 13. If $j < n$ then set $i = i + 1$ and go to Step 10.
Step 14. Set $i = 1$.

Step 15. Set $j = i + length - 1$.
Step 16. If the move FLIP(i, j, π) reduces the cost then perform the move FLIP(i, j, π) and set *improve = true*.
Step 17. If $j < n$ then set $i = i + 1$ and go to Step 15.
Step 18. If $length \leq n - 3$ then set $length = length + 1$ and go to Step 3 (Moves for lengths $n - 2$ and $n - 1$ are simulated by earlier moves).
Step 19. If *improve = true* then go to Step 2.
Step 20. Output the permutation π and stop.

The complexity of LESS can be assessed as follows. In Step 5 of LESS, a search is conducted for the best position k for the block B_{ij}. There are $O(n - length)$ distinct possible values for k. For each such value x, a simulation of the move TRANSFER(i, j, k, π) must be performed in order to calculate the reduction in cost *gain*. If one assumes that each node has a bounded number of nets connected to it, and each net connects a bounded number of nodes, then the number of nets $m = O(n)$ and the length of each net in a given permutation can be computed in constant time. Thus, the complexity of one simulation of the move TRANSFER is at most $O(n)$, since at most $O(n)$ nodes change positions and the length of at most $O(n)$ nets must be updated. Consequently, the complexity of Step 5 is $O(n(n - length))$. Hence, the complexity of the loop embodied by Steps 4–7 is $O(n(n - length)^2)$. A similar analysis shows that the complexity of the loops embodied by Steps 10–13 and by Steps 15–17 are ($O(n(n - length)^2)$ and $O(n(n - length))$, respectively. Finally, the complexity of one phase of LESS (Steps 3–18) is $O\sum_{i=1}^{n} n(n - i)^2) = O(n^4)$. LESS executes few phases (In our experiments, the number of phases never exceeded 15) before it stops, so that in fact the overall complexity of LESS is also $O(n^4)$.

4 EXPERIMENTAL RESULTS AND FINE TUNING OF LESS

All our experiments were performed on a Sun Sparc 1+ workstation. LESS was first tested on small circuits collected from the literature. In some cases, the nets have heavy integer weights. Since LESS assumes that all the nets have unit weights, every net of weight i was replaced by i nets identical to it. The results are summarized in Table I. The initial permutation was generated randomly using the following procedure:

Step 1. Set $\pi(i) = i$ for each $1 \leq i \leq n$.
Step 2. Repeat $3n$ times: randomly choose two distinct integers i and j from the set $\{1, 2, \ldots, n\}$ and then swap $\pi(i)$ and $\pi(j)$.
Step 3. Return the permutation π.

TABLE I
Performance of LESS on Small Circuits Collected From the Literature

Circuit	Nodes	Nets	Time(s)	No. Phases	Cost	Best
fig1.2 [13]	5	131	0.3	2	243	243
fig1.4 [13]	5	31	0.1	2	43	43
fig9a [12]	6	50	0.2	2	78	78
fig8 [11]	9	21	0.3	2	50	50
Data III [19]	15	18	2.1	3	65	71
Data V [19]	29	37	33.5	3	220	254
Data VI [19]	48	48	175.4	3	337	411
fig10 [7]	85	96	2209.1	4	1009	1470

The results indicate the high quality solutions of LESS in comparison with the best previously known solutions which are given in the last column of Table I. The running time of LESS in seconds is given for each circuit in Table I. For the data collected from the literature in Table I, the running time is not available except for circuit fig10 [7] for which the a cost of 1470 was obtained by an algorithm described in [7] in 6.1 seconds.

LESS was then used on two larger circuits in order to get a better idea about its running time as well as its performance. The performance was estimated by comparing the results of LESS to the results of the SE algorithm [10] on these two circuits. In [10], the SE algorithm has been empirically shown to be more effective than simulated annealing [17] and Kang's constructive algorithm [1]. Table II shows for each of the two circuits, the number of nodes and nets, the cost of the solution found by LESS, the cost of the solution found by SE, the execution time of LESS in hours, and the number of phases of LESS. To make our comparison meaningful, the SE algorithm was given the same running time as LESS. For both circuits, the solution found by LESS is better than the solution found by SE. Unfortunately, it took LESS fourteen and a half hours to solve circuit CKT2.

While the results of LESS are excellent in terms of the quality of the solutions obtained, the running time of LESS may be impractical especially if LESS is to be used to solve much larger circuits. Therefore,

TABLE II
Performance of LESS on Circuits of Moderate Sizes

Circuit	Nodes	Nets	Cost	SE Cost	Time (h)	No. Phases
CKT1	60	60	608	696	0.25	4
CKT2	183	165	1230	1262	14.30	5

in order to make LESS more practical, its running time must be reduced while maintaining the quality of its solutions. One way of achieving that is by limiting LESS to execute only the moves that are likely to reduce the cost by a considerable amount. Notice that each phase of LESS can be considered as the application of a sequence of $n - 2$ iterative improvement algorithms parameterized by the variable *length*. The i-th improvement algorithm tries to improve the cost by performing the moves TRANSFER, TRANSFER_FLIP, or FLIP on every block of length i. In a phase of LESS, call length i *useful* if the i-th improvement algorithm was capable of reducing the solution cost. To understand which of the $n - 2$ lengths considered during a phase are useful, LESS was programmed to output the cost after every length, i.e., the cost of the current solution is written into an output file before LESS executes Step 18. It was observed that very few lengths are useful in a phase of LESS. For example, the lengths which were useful in the first phase of LESS while working on circuit CKT2 are: 1–10, 15, 17–20, 28–29, 57–58, 155, 178, 180. It is clear that there are large gaps between periods of useful lengths. For instance, the lengths 59 through 154 were useless. Because no improvement occurs during periods of useless lengths, the time spent on trying to execute moves at these lengths is wasted. Moreover, notice that there are more useless lengths than useful ones: Out of 181 lengths, only 22 lengths are useful. This means that much of the time spent by LESS is a wasted time. The sequence of costs corresponding to the above useful lengths is: (1, 7729), (2, 3900), (3, 3276), (4, 2846), (5, 2617), (6, 2478), (7, 2309), (8, 2268), (9, 2257), (10, 2255), (15, 2254), (17, 2112), (18, 1839), (19, 1790), (20, 1779), (28, 1725), (29, 1648), (57, 1633), (58, 1576), (155, 1553), (178, 1551), (180, 1550). For convenience, this sequence is given in the form (useful length, cost). The cost of the initial solution is 15106. One should notice now that the majority of useful lengths are short. Furthermore, the largest reductions in cost occurs at short lengths. Therefore, much of the quality of the final solution is retained if only the first few lengths were considered in LESS. This can be achieved by changing Step 18 in LESS to:

New_Step 18. if *length* $\leq n/\alpha$ then set *length* = *length* + 1 and go to Step 3,

where $\alpha > 1$ is a parameter chosen by the user. The computation time can be reduced by choosing a large value for α. Call the modified algorithm LESS(α). LESS(10) was used on circuits CKT1 and CKT2, and its results were compared to those of LESS in Table III. It can be seen that for each of the two circuits CKT1 and CKT2, the quality of the solution of LESS(10) is slightly worse than the solution of LESS. However, LESS(10) is much faster than LESS.

It is also possible to improve the running time of LESS by cutting down on the computation of Steps 5 and 11. Here, it is shown how Step 5 can be improved. Step 11 can be modified similarly. The computation of Step 5 consists of a search for a position $k \notin [i - 1, j]$ such that if the block B_{ij} is transferred to this position, then the reduction in the cost of the permutation is maximum. Therefore, if block B_{ij} has length l, then $O(n - l)$ positions are tested. If the set of candidate positions is restricted to a small subset of the set of all possible positions, then the speed of the algorithm can be improved considerably. The first observation is that the reduction in cost due to the transfer of block B_{ij} to a new position k should not be very different from the reduction due to the transfer of B_{ij} to a position adjacent to k. The second observation is that as the candidate positions are tested from left to right (If $k_1 < k_2$, then we say that position k_1 is to the left of k_2), the reduction in cost changes noticeably at critical positions. To illustrate this second observation, suppose all the nodes of the circuit except one node (say node y) has been arranged in some order. To insert node y in this order, there is a position k such that if node y is squeezed in this position, then the cost of the resulting permutation is no more than the cost of the permutation which results from squeezing y in any other position. By squeezing y in position k we mean that the nodes in positions $l > k$ are shifted one unit to the right

TABLE III
Performance of LESS(10) on Circuits of Moderate Sizes

			Cost		Time (h)		No. Phases	
Circuit	Nodes	Nets	LESS(10)	LESS	LESS(10)	LESS	LESS(10)	LESS
CKT1	60	60	678	608	0.03	0.25	3	4
CKT2	183	165	1537	1230	2.06	14.30	4	5

TABLE IV
Comparison of FAST($n/20$) and SE Starting From a Random Initial Permutation

Circuit	Nodes	Nets	Cost		Time (h)	No. Phases
			FAST($n/20$)	SE	FAST($n/20$)	FAST($n/20$)
CKT1	60	60	621	714	0.03	4
CKT2	183	165	1542	1318	0.09	5
CKT3	200	300	9155	9922	1.89	4
CKT4	286	307	4031	4056	0.56	6
CKT5	469	451	6450	6658	1.11	7
CKT6	800	684	38612	34068	12.78	12
CKT7	3060	3123	304626	213280	209.38	15

and then y is put in position $k + 1$. Thus, the range of k is $[0, n]$. Let C be the cost of the current permutation of all the nodes but node y, which ignores the length of the nets connected to y (Given a permutation π, the length of a net N_j is equal to $h(\pi, j) - l(\pi, j)$). Let $CUT(k)$ be the set of nets that are not connected to y, and that have nodes in positions less or equal to k as well as in positions greater than k. Let c_k be the number of nets in $CUT(k)$. Let l_k be the total length of the nets connected to y when y is squeezed in position k. Then, the cost of the resulting permutation of all the nodes is equal to $C + l_k + c_k$. The term c_k is the squeeze factor which is due to the fact that the length of every net in $CUT(k)$ increases by 1 if node y is squeezed in the k-th position. It is clear now that the best position for node y in the existing order of the other nodes is the one that minimizes $l_k + c_k$. If variations in c_k are negligible, then the best position k for node y is the one that minimizes l_k. Now notice that as k increases from 0 to n, l_k first decreases steadily up to a certain position m, it then stays at the same value until some position M (M can be the same as m) is reached, and it increases steadily after that. Such a function l_k of k is called a *convex* function. It can be verified that the sum of convex functions is convex. Therefore, to show that l_k is convex, it suffices to show that the length of every net connected to y is convex. Let w_k be the length of some net N_j connected to y if y is inserted in position k. Let m_j (M_j) be the minimum and maximum position of the nodes other than y in N_j in the existing order. Clearly, we have:

$$w_k = \begin{cases} M_j - k & \text{if } k < m_j \\ M_j - m_j + 1 & \text{if } m_j \le k \le M_j \\ k - m_j + 1 & \text{if } k > M_j \end{cases}$$

Therefore, w_k is a convex function of k. The minimum of w_k is achieved at $k = m_j$ and $k = M_j$. Therefore, if Y is the set of all nets connected to y and $S = \{m_j, M_j: N_j \in Y\}$, then the minimum of l_k must be achieved at some $k \in S$. In general, let Y_{ij} be the set of all nets connected to nodes in the block B_{ij}, and let $S_{ij} = \{m_x, M_x: N_x \in Y_{ij}\} - \{i - 1, i, i + 1, \ldots, j\}$. Now, Step 5 can be restricted to search for a best $k \in S_{ij}$. Step 11 can be similarly modified.

Call FAST(α) the algorithm LESS(α) in which Steps 5 and 11 has been modified as discussed above. FAST(α) was used on a number of circuits. For every one of these circuits, α was set equal to $n/20$ so that the maximum length of a block is no more than 20. The results of FAST($n/20$) were compared to those obtained by the SE algorithm [10]. The same initial random permutation was used by both algorithms. Table IV shows the results. The time shown in Table IV is the running time of FAST($n/20$) in hours. The SE algorithm was given the same running time. A comparison of the running times of LESS, LESS(10), and FAST($n/20$) on circuit CKT2, shows that FAST($n/20$) is indeed faster than both LESS and LESS(10). The solutions found by FAST($n/20$) are comparable in quality to the corresponding solutions found by the SE algorithm. This illustrates that even under the restriction imposed on the basic algorithm, it is still possible to obtain good quality solutions.

Another possibility for improving the running time of the basic algorithm is to start with a good initial permutation. First, the cost of a good initial permutation is expected to be less than the cost of a random permutation, so that fewer iterations may be needed by the algorithm. Second, in a good permutation, nodes that are heavily connected are expected to be close to each other. Consequently, the moves TRANSFER and TRANSFER_FLIP should be less time consuming (The computations of TRANSFER(i, j, k, π) and TRANSFER_FLIP(i, j, k, π) are proportional to $j - k$ if $k < i$ and to $k - i$ if $k > j$). Clearly however, starting from a constructive initial permutation need not lead to better solutions.

TABLE V
Comparison of FAST($n/20$) and SE Starting From an Initial Permutation Generated by CLO

			Cost		Time (h)	No. Phases
Circuit	Nodes	Nets	FAST($n/20$)	SE	FAST($n/20$)	FAST($n/20$)
CKT1	60	60	621	652	0.04	4
CKT2	183	165	1640	1670	0.07	4
CKT3	200	300	9119	9410	2.53	5
CKT4	286	307	4153	4086	0.31	4
CKT5	469	451	6346	7216	0.74	5
CKT6	800	684	34640	36393	7.28	7
CKT7	3060	3123	156300	197054	76.52	11

The heuristic CLO [10] was used to generate initial permutations to both FAST($n/20$) and SE. The new results of both algorithms are shown in Table V. Again, SE was given the same running time as FAST($n/20$). A comparison of the results in Tables IV and V shows that for most circuits it was advantageous to start with a good initial permutation. The improved performance is obvious for larger circuits. For circuit CKT7 for example, the solution of Fast($n/20$) starting from the initial permutation generated by CLO has about half the cost of the solution of Fast($n/20$) starting from a random initial permutation. In addition, for circuit CKT7, the use of a good initial permutation made FAST($n/20$) about three times as fast. The results in Tables 4 and 5 show a dependency on the initial permutation. However, the quality of the final solution of the original algorithm LESS is much less sensitive to the initial permutation as is indicated by the empirical results. The restriction on blocks to be of short length and the limitations imposed on Steps 5 and 11 in FAST($n/20$) enhance the sensitivity of the algorithm to the initial permutation.

In the appendix, we give a procedure for the construction of test circuits for which the optimal cost is known. The input to the construction procedure consists of the desired number of nodes n, the desired number of nets m, and a desired bound $B > 2$ on the maximum number of nodes that can be connected by a single net. The construction procedure outputs the constructed circuit along with its optimal cost. Using this procedure, a number of circuits were generated. Then, FAST($n/20$) and the SE were tried on these circuits using a random initial permutation and an initial permutation generated by CLO. The results obtained by initially using a random permutation and the permutation generated by CLO are shown in Table VI and Table VII, respectively. A comparison of the results in Tables VI and VII reveals that better results can be achieved in a much shorter time if a good initial permutation is used. This is true for both FAST($n/20$) and the SE algorithm. When a permutation generated by CLO is used as an initial permutation, FAST($n/20$) results are close to being optimal in Table VII. The speedup over the use of a random initial permutation is also clear. For circuit C8 for example, FAST($n/20$) starting from a good initial permutation generated by CLO, achieved a solution whose cost is less than 1/3 the cost obtained when a random initial permutation is used with a speedup of 13.71. As can be observed in Table VI, when the initial permutation is random, the solution of FAST($n/20$) becomes more and more away from the optimal solution as the size of the circuit becomes larger. This is mostly due to limiting the computation to blocks of length less or equal to 20. When the size

TABLE VI
Comparison of FAST($n/20$) and SE on Artificial Circuits Using a Random Initial Permutation

			Cost			Time (h)	No. Phases
Circuit	Nodes	Nets	FAST($n/20$)	SE	Optimal	FAST($n/20$)	FAST($n/20$)
C1	60	75	372	372	372	0.03	5
C2	100	125	747	1027	669	0.05	4
C3	200	239	1685	2545	1240	0.14	5
C4	400	421	3841	5961	2263	0.7	8
C5	600	680	7225	15670	3694	1.49	8
C6	800	841	9253	26054	4538	2.05	8
C7	1000	1250	22584	41203	6869	5.16	8
C8	1500	1610	37633	82016	8959	10.69	10

TABLE VII
Comparison of FAST($n/20$) and SE on Artificial Circuits Using an Initial Permutation Generated by CLO

			Cost			Time (h)	No. Phases
Circuit	Nodes	Nets	FAST($n/20$)	SE	Optimal	FAST($n/20$)	FAST($n/20$)
C1	60	75	372	516	372	0.01	3
C2	100	125	865	983	669	0.04	4
C3	200	239	1359	1378	1240	0.06	3
C4	400	421	2446	2661	2263	0.14	4
C5	600	680	3887	4558	3694	0.25	4
C6	800	841	4964	6083	4538	0.25	3
C7	1000	1250	7925	8187	6869	0.43	3
C8	1500	1610	10006	11476	8959	0.78	4

of the circuit is small (circuit C1 for example), an optimal solution can still be achieved. However, for larger circuits, length 20 becomes too small in comparison with the number of nodes. Consequently, FAST($n/20$) becomes more susceptible to be trapped in local minima. To avoid this situation, longer blocks need to be considered. This certainly will increase the computation time enormously since the majority of possible block lengths are useless as it was mentioned earlier in this section. When longer blocks are allowed, the algorithm may perform many useless computations before it reaches a useful length during each phase.

Note that SE was capable of obtaining better solutions than FAST($n/20$) for some cases in Tables IV and V, but the results of SE are consistently worse than the results of FAST($n/20$) in Tables VI and VII. While it is not possible to give a conclusive justification of this observation, it seems that SE does not perform as well on the artificial circuits used in Tables VI and VII. This may be partly due to the fact that SE uses only two-exchange moves.

The above results show that FAST($n/20$) is effective and more practical and faster than LESS. Therefore, it is natural to check the performance of FAST($n/20$) on the same circuits in Table 1, which were first used to test the effectiveness of LESS. The results of FAST($n/20$) on these circuits are summarized in Table VIII (The last column shows the best previously known solutions). These results are comparable to the one obtained by LESS in Table I, but they were obtained in a much shorter time. Also, for Data V, FAST($n/20$) obtained a slightly better solution than LESS.

To illustrate that FAST(α) can indeed produce near-optimal solutions if larger blocks are considered (e.g., a small value of α is used), FAST(2) was used on the smaller circuits in Tables VI and VII (up to 600 nodes). For larger circuits, FAST(2) uses more computation time than we can afford on a Sun Sparc 1+ workstation. Again, two different choices for the initial permutation were used: a random one, and a constructive one generated by CLO. The results are shown in Table IX. It can be seen that near-optimal solutions can be achieved. In fact optimal solutions were generated for circuits C1, C2, and C5 regardless of the initial permutation. The worst cost was generated for circuit C4 when the initial permutation was randomly generated. Nevertheless, even in this case, the cost of the solution generated is within 10% of the optimal cost. The running time and the cost were consistently less when the initial permutation was generated by CLO. The saving in running time can be significant. For example, for circuit C5, FAST(2) was five times faster when the permutation generated by CLO was used as the initial permutation.

TABLE VIII
Performance of FAST($n/20$) on Small Circuits Collected From the Literature

Circuit	Nodes	Nets	Time(s)	No. Phases	Cost	Best
fig1.2 [13]	5	131	0.3	2	243	243
fig1.4 [13]	5	31	0.1	2	43	43
fig9a [12]	6	50	0.1	2	78	78
fig8 [11]	9	21	0.3	2	50	50
Data III [19]	15	18	1.1	3	65	71
Data V [19]	29	37	13.8	3	217	254
Data VI [19]	48	48	39.0	4	341	411
fig10 [7]	85	96	301.6	7	1013	1470

TABLE IX
Empirical Evidence of the Near-Optimality of FAST(2) Solutions

	Time (h)		Cost		
Circuit	Random	CLO	Random	CLO	Optimal
C1	0.05	0.02	372	372	372
C2	0.23	0.11	669	669	669
C3	1.46	0.86	1244	1242	1240
C4	16.57	5.87	2479	2305	2263
C5	101.27	20.29	3694	3694	3694

5 CONCLUDING REMARKS

The notion of a "block" in a permutation is the fundamental concept used in our linear placement algorithm. A good block should be a set of consecutive nodes which form a cluster (i.e., a set of heavily interconnected nodes). If there is a simple criterion for determining whether a sequence of consecutive nodes is a good block, then LESS can be made much faster. The lack of such criterion, forces us to consider all $(n^2 + n)/2$ subsequences of consecutive nodes in a permutation as blocks. As part of future work, we intend to develop such a criterion.

Even though part of the computation time is unproductive, LESS can be modified into the effective algorithm FAST(α). The algorithm FAST(α) restricts the length of a block to be at most n/α. Thus, the computation time is inversely proportional to α. However, the cost of the final solution is proportional to α. Consequently, α must be chosen appropriately so that the computation time is sharply reduced without damaging the quality of the final solution.

Our experiments indicate that only a small fraction of possible block lengths are useful. The smallest lengths (1–10) were most of the time useful. However, larger useful lengths (>10) are separated by gaps of many useless lengths. These gaps are traps that prevent heuristics from achieving good solutions. LESS had to step over many of these traps in order to discover the useful lengths. The existence of these gaps may be an indication that probabilistic algorithms such as simulated annealing [17] and stochastic evolution [18] may need a long time to generate near-optimal solutions if only simple moves (e.g., transpositions) are used.

Acknowledgment

The authors would like to thank the unknown reviewers for their helpful comments which improved this paper.

References

[1] S. Kang, "Linear Ordering and Application to Placement," *Proc. 20th Design Automation Conference,* pp. 457–464, 1983.

[2] J. Cohoon and S. Sahni, "Heuristics for the Board Permutation Problem," *Proc. Int. Conf. Computer-Aided Design,* pp. 81–83, 1983.

[3] I. Cederbaum, "Optimal Backboard Ordering Through the Shortest Path Algorithm," *IEEE Trans. Circuits and Systems,* Vol. CAS-21, pp. 626–632, 1974.

[4] S. Goto, I. Cederbaum, and B. Ting, "Suboptimum Solution of the Back-Board Ordering with Channel Constraint," *IEEE Trans. Circuits and Systems,* Vol. CAS-24, pp. 645–652, 1977.

[5] A. Sangiovanni-Vincentelli and M. Santomauro, "A Heuristic Guided Algorithm for Optimal Backboard Ordering," *Proc. 13th Annual Allerton Conference on Circuit and System Theory,* pp. 916–921, Oct. 1975.

[6] G. Persky, "PRO—An Automatic String Placement Program for Polycell Layout," *Proc. 13th Design Automation Conference,* pp. 417–424, 1976.

[7] S. Yamada, H. Okude, and T. Kasai, "A Hierarchical Algorithm for One-Dimensional Gate Assignment Based on Contraction of Nets," *IEEE Trans. Computer-Aided Design,* Vol. CAD-8, No. 6, pp. 622–629, June 1989.

[8] H. Cho and C. Kyung, "A Heuristic Cell Placement Algorithm Using Constrained Multistage Graph Model," *IEEE Trans. Computer-Aided Design,* Vol. 7, No. 11, pp. 1205–1214, November 1988.

[9] S. Chowdhury, "Analytical Approaches to the Combinatorial Optimization in Linear Placement," *IEEE Trans. Computer-Aided Design,* Vol. 8, No. 6, pp. 630–639, June 1989.

[10] Y. Saab and V. Rao, "Linear Ordering by Stochastic Evolution," in *Proc. 4th CSI/IEEE Int. Symp. on VLSI Design,* New Delhi, India, Jan. 1991, pp. 130–135.

[11] D. Schuler and E. Ulrich, "Clustering and Linear Placement," *Proc. 9th Design Automation Conference,* pp. 57–62, 1972.

[12] C. Cheng, "Linear Placement Algorithms and Applications to VLSI Design," *Networks,* Vol. 17, pp. 439–464, 1987.

[13] D. Adolphson and T. Hu, "Optimal Linear Ordering," *SIAM J. Appl. Math.,* Vol. 25, No. 3, November 1973.

[14] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness.* New York, NY: W.H. Freeman and Company, 1979.

[15] L. Ford and D. Fulkerson, *Flows in Networks.* Princeton, NJ: Princeton University Press, 1962.

[16] S. Goto, "An Efficient Algorithm for the Two-Dimensional Placement Problem in Electrical Circuit Layout," *IEEE Trans. Circuits and Systems,* Vol. CAS-28, pp. 12–18, January 1981.

[17] S. Kirkpatrick, C. Gelatt, and M. Vecchi, "Optimization by Simulated Annealing," *Science,* Vol. 220, pp. 671–680, May 1983.

[18] Y. Saab and V. Rao, "Combinatorial Optimization by Stochastic Evolution," *IEEE Trans. Computer-Aided Design,* Vol. 10, No. 4, pp. 525–535, April 1991.

[19] Y. Hong, K. Park, and M. Kim, "A Heuristic Algorithm for Ordering the Columns in One-Dimensional Logic Arrays," *IEEE Trans. Computer-Aided Design,* Vol. 8, No. 5, pp. 547–562, May 1989.

APPENDIX

Construction of Test Cases for Linear Placement Heuristics

A method for the construction of test cases for the evaluation of linear placement heuristics is presented. The circuits generated by this method are by no means claimed to model circuits encountered in real applications. Furthermore, these circuits are not guaranteed to be connected†. Nevertheless, these

†The method can be easily modified to guarantee the connectedness of the generated circuits.

circuits can be used to estimate the effectiveness of a heuristic. In particular, unless a heuristic is poor, it should perform reasonably well on these circuits.

The idea behind our construction method is very simple: If there is a permutation π of the nodes of a circuit such that the cost of π is equal to a lower bound on the optimal cost, then this permutation is an optimal solution to the linear placement problem. Clearly, the length of any net in any permutation cannot be less by more than one than the number of nodes connected by this net. Therefore, if a circuit has m nets, where the number of nodes connected by net N_j is x_j, then the cost $L(\pi)$ of any permutation π must satisfy

$$L(\pi) \geq \sum_{j=1}^{m} (x_j - 1). \qquad (*)$$

The input to the algorithm consists of the number of nodes n, the number of nets m, and a bound $B > 2$ on the maximum number of nodes that can be connected by a single net. The algorithm generate nets connecting at most B nodes. However, the algorithm can be easily modified so that a fraction of nets connect only 2 nodes each, 3 nodes each, . . . , etc. The algorithm begins with a random permutation π and constructs the nets so that equality holds in (*). Random permutations are generated as described in Section 4 of this paper. Here is the outline of this algorithm:

Step 1. Read n, m, and B.
Step 2. Generate a random permutation π of the set of nodes $\{1, 2, \ldots, n\}$.
Step 3. Set $k = 1$ and $C_o = 0$.
Step 4. Let i be a random integer in the interval $[1, n]$. Let $j \neq i$ be a random integer in the interval $[1, n] \cap [i - B + 1, i + B - 1]$. Suppose without loss of generality that $i < j$. Construct the net $N_k = \{\pi(l): i \leq l \leq j\}$. Set $C_o = C_o + j - i$.
Step 5. Increment k. If $k \leq m$ then go to Step 4.
Step 5. Output the circuit and the optimal cost C_o.

Clearly, all the nets are constructed such that their nodes are consecutive in the random permutation π. Therefore, for this permutation π, equality holds in (*). Consequently C_o which at the end of the algorithm is the cost of π, is indeed the optimal cost for the constructed circuit. It should be mentioned here that the initial random permutation π need not be the only optimal solution.

Biographies

YOUSSEF G. SAAB received the B.S. degree in computer engineering and the M.S. and the Ph.D. degrees in electrical engineering from the University of Illinois at Urbana-Champaign, Urbana, IL, in 1986, 1988, and 1990, respectively. He is currently an assistant professor in the department of computer science at the University of Missouri-Columbia. From January 1986 to July 1990, he was a research assistant at the Coordinated Science Laboratory, Urbana, IL. His research interests include computer-aided design and layout of VLSI circuits, combinatorial optimization, computational geometry, and graph algorithms. Dr. Saab is a member of IEEE, ACM, Tau Beta Pi, Eta Kappa Nu, Phi Kappa Phi, and the Golden Key National Honor Society.

CHENG-HUA CHENG received the B.E. degree in computer engineering from the National Chiao-Tung University, Hsinchu, Taiwan, in 1986, and the M.S. degree in computer science from the University of Missouri-Columbia in 1991. He is currently working with Behavior Design Corporation, Hsinchu, Taiwan. From 1988 to 1989, he worked for the development of CAS-200 (a computer-aided design system for analog circuit layout) at a private company in Taiwan. His interests include database systems, CAD tools for VLSI design, computer graphics, and computational geometry.