OPTIMAL LINEAR ARRANGEMENT OF CIRCUIT COMPONENTS*

Jayaram Bhasker and Sartaj Sahni

University of Minnesota

ABSTRACT

We study the problem of arranging circuit components on a straight line so as to minimize the total wire length needed to realize the inter component nets. Branch-and-bound and dynamic programming algorithms that find optimal solutions are presented. In addition, heuristic approaches including some that employ the Monte Carlo method are developed and an experimental evaluation provided.

Keywords and Phrases

Optimal linear arrangement, complexity, branch-and-bound, dynamic programming, heuristic, Monte Carlo method.

* This research was supported in part by the National Science Foundation under grant MCS-83-05567.

1 INTRODUCTION

In the *Hypergraph Optimal Linear Arrangement* (HOLA) problem, we are given a hypergraph, $H = (V, S)$, and a set, $W = \{w_1, w_2, ..., w_k\}$, of weights. $V = \{1, 2, ..., n\}$ is a set of vertices and $S = \{S_1, S_2, ..., S_k\}$ is a collection of subsets of V. A *linear arrangement* of the vertices, V, is a permutation $\sigma = \sigma(1)\ \sigma(2)\ ....\ \sigma(n)$ of $(1, 2, ..., n)$. Vertex i is assigned the $\sigma(i)$th position in the linear arrangement. The cost of a permutation σ is:

$$cost(\sigma) = \sum_{i=1}^{k} w_i \max_{\substack{q,l \in S_i \\ q<l}} \{ \, |\sigma(q) - \sigma(l)| \, \} \tag{1}$$

The objective of the HOLA problem is to obtain a σ with minimum cost. The HOLA problem with $w_i = 1$ models several circuit arrangement problems. For example, the problem of ordering the boards in a backplane to minimize total wire length is the HOLA problem with vertices representing boards and the S_i´s representing nets. The problem of reordering columns in a column structured gate array to minimize horizontal wire length is similarly modeled.

A special case of the HOLA problem arises when the hypergraph H is actually a graph. This is the case when the S_i´s are all different and each S_i contains exactly two vertices. We shall refer to this special case as the *Graph Optimal Linear Arrangement* (GOLA) problem. GOLA with edge weights equal to 1 is referred to as the *Optimal Linear Arrangement* (OLA) problem. Equation (1) may be restated for the case of graphs. Given any graph, $G = (V, E)$, with $w(i,j)$ denoting the weight of edge $(i, j) \in E$, we wish to find a permutation σ that minimizes:

$$cost(\sigma) = \sum_{(i,j) \in E} w(i,j) \, |\sigma(i) - \sigma(j)| \tag{2}$$

While this version of HOLA is not of much interest in circuit design (as it corresponds to the case when each net covers only two components or pins), it is of interest in complexity circles as several interesting results are available for this problem. For example, the OLA problem is known to be NP-hard ([GARE76] and [EVEN75]). Since OLA instances form a subset of HOLA instances, HOLA is also NP-hard. The GOLA problem restricted to the case of rooted trees is solvable in O($n$log$n$) (where $n = |V|$) time ([ADOL73]). The OLA problem, when restricted to undirected trees, is solvable in $O(n^{2.2})$ time ([SHIL79]).

The GOLA problem (i.e., weighted OLA) has been studied in [ADOL73]. Heuristics for the HOLA problem appear in [SCHU72] and [HANA72], and for the HOLA problem with unit weights ($w_i = 1$) in [KANG83].

In this paper, we are primarily concerned with the HOLA problem. We first show that obtaining k-absolute approximate or ε-approximate solutions ([HORO78] and [GARE79]) for the HOLA problem is NP-hard (actually, we show this for the restricted version, OLA). Next, we develop algorithms to obtain optimal arrangements. Since the optimal arrangement problem is

NP-hard, we do not expect to solve this problem in polynomial time. Both our algorithms are of non polynomial complexity. Hence, their application is limited to very small instances. One algorithm uses the branch-and-bound technique and the other uses dynamic programming. Finally, we explore heuristic approaches including the Monte Carlo method. Our experimental results show that the Monte Carlo method as well as another heuristic we develop, can be used to obtain better solutions than those obtained by earlier proposed algorithms ([KANG83] and [SCHU72]).

2 NP-HARD APPROXIMATION PROBLEMS

Definitions: [HORO78] Let P be a problem and let I be any instance of P. Let $f^*(I)$ be the value of an optimal solution to I and let $f'(I)$ be the value of some feasible solution (a feasible solution is any solution that satisfies the constraints for P). A feasible solution is a *k-absolute approximate solution* iff $|f'(I) - f^*(I)| \leq k$. It is an ε-*approximate solution* iff $|f'(I) - f^*(I)| / f^*(I) \leq \varepsilon$. The *k-absolute approximation problem* (ε-*approximation problem*) is that of finding k-absolute approximate (ε-approximate) solutions. □

Theorem 2.1 shows that both the k-absolute approximation and ε-approximation problems for OLA are NP-hard. Since OLA is a restriction of HOLA to graphs with unit edge weights, this theorem also establishes the NP-hardness of these approximation problems for HOLA.

Theorem 2.1: The k-absolute approximation problem and the ε-approximation problem for OLA are NP-hard.

Proof: We first establish this for the case of the k-absolute approximation problem. This is done by showing how a polynomial time algorithm, A, for the k-absolute approximation problem can be used to obtain true optimal solutions for OLA.

Let G be any instance of OLA. From G, construct a new instance G' that is comprised of $k + 1$ copies of G (Figure 2.1). From any k-absolute approximate solution for G', one can easily extract an optimal solution for G. This is done as follows. Suppose the obtained k-absolute approximate solution is σ. Let σ_i denote the order in which the vertices of the ith copy of G appear in σ. Clearly, $cost(\sigma) \geq \sum_{i=1}^{k+1} cost(\sigma_i)$, where $cost(\sigma_i)$ is computed as it would be for a single copy of G using the permutation σ_i.

Let τ be an optimal ordering for the vertices in G. An optimal ordering for G' is obtained by ordering the vertices in each copy of G according to τ and then concatenating these together. This ordering has a cost of $(k + 1)\, cost(\tau)$.

If σ is a k-absolute approximate solution for G', then $cost(\sigma) - (k+1)\, cost(\tau) \leq k$. Hence, $\sum_{i=1}^{k+1} cost(\sigma_i) - (k+1)\, cost(\tau) \leq k$. Since $cost$ is an integer valued function, this inequality can hold only if at least one of the σ_i's has a cost equal to $cost(\tau)$ (otherwise, $cost(\sigma_i) \geq cost(\tau) + 1$

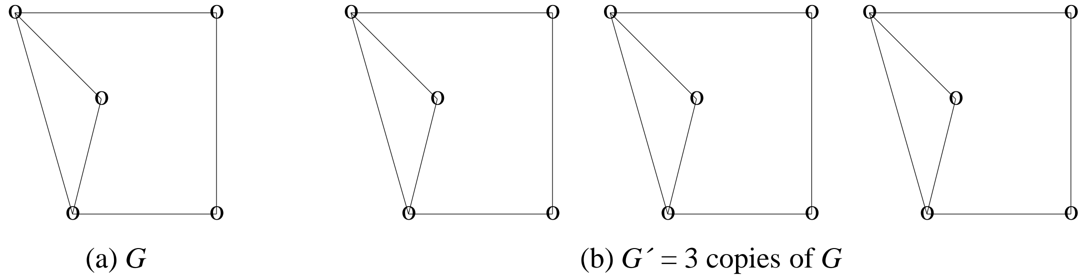

Figure 2.1: Example G and G' for the case $k = 2$.

for every i and $\sum_{i=1}^{k+1} cost(\sigma_i) \geq (k+1)\, cost(\tau) + k + 1)$. Hence, at least one of the σ_i´s is optimal for G.

So, an optimal ordering for G may be obtained by executing the following steps:

a) Construct G' by making $k + 1$ copies of G.

b) Use the k-absolute approximation algorithm on G´ to obtain σ.

c) Extract the $k + 1$ permutations $\sigma_1, \sigma_2, ..., \sigma_{k+1}$ from σ.

d) Compute the cost of each σ_i for G.

e) Pick the σ_i with least cost. This is the optimal ordering for the vertices in G.

Steps a), c), d), and e) can be done in polynomial time. If Step b) can be done in polynomial time, then OLA is polynomially solvable. Since OLA is NP-hard, Step b) is also NP-hard.

The proof for the NP-hardness of the ε-approximation problem is similar. Instead of using $k + 1$ copies of G, we use $\lceil (1 + \varepsilon)\, n\, (n^2 - 1)/6 \rceil + 1$ copies. □

3 OPTIMAL SOLUTIONS

In this section, we develop two algorithms to obtain optimal solutions to HOLA. The asymptotic complexity of each is such that these algorithms are practical only for small n.

3.1 Branch-and-Bound

Branch-and-bound (see [HORO78], for e.g.) is a systematic way to search the solution space of a problem. This search avoids examining all elements of this space by using bounding functions. The effectiveness of branch-and-bound is critically dependent on the quality of the bounding functions in use.

For the case of HOLA, the solution space is a permutation space (i.e., it consists of all permutations of the n vertices $\{1, 2, ..., n\}$). The size of this space is $n!$. Searching the entire space for an optimal solution is impractical for large n. The use of good bounding functions (in this case, lower bounds on the cost, $c*$, of the optimal arrangement) can significantly reduce the number of elements in the solution space that get examined during the branch-and-bound search. This subsection is devoted to the development of good bounding functions for HOLA and GOLA.

At the time a bounding function is used, some of the vertices in the hypergraph, $H = (V, S)$, have been placed at the left end of the linear arrangement. Let P denote this subset. A lower bound, $L(P)$, on the cost of any linear arrangement of V that has the given ordering of P at its left end is obtained by summing the three components: $L^A(P)$, $L^B(P)$, and $L^C(P)$. To define these three components, we partition S into three sets A, B, and C as below:

A = sets that are contained only in $P = \{i \mid S_i \subseteq P\}$

B = sets partially contained in P

$= \{i \mid S_i \cap P \neq \emptyset \text{ and } S_i \not\subseteq P\}$

C = sets not represented in P

$= \{1, 2, ..., k\} - \{A \cup B\}$

1. $L^A(P) = \sum_{i \in A} cost(S_i)$,

where $cost(S_i) = w_i$ * (maximum distance between two vertices in S_i). Since $S_i \subseteq P$, the position of all vertices in S_i is known and $cost(S_i)$ can be computed.

2. $L^B(P) = \sum_{i \in B} cost(S_i)$

The position of some of the elements in S_i is not known at this time. Let $d_i = |P| + |\{i \mid i \in S_i \text{ and } i \notin P\}|$. Let f_i be the position of the leftmost vertex in $S_i \cap P$. Clearly, $cost(S_i) = w_i * (d_i - f_i)$ is a lower bound on the cost of S_i in any completion of the given initial arrangement.

3. $L^C(P) = \sum_{i \in C} cost(S_i)$

The value of $L^C(P)$ to use is obtained by using one of the lower bounds discussed in Sections 3.1.1 - 3.1.3.

3.1.1 Lower bounds for C ... GOLA

$\sum_{i \in C} w_i$ is a trivial lower bound when the hypergraph is actually a graph. Let us call this lower bound $lb1$. Adolphson and Hu [ADOL73] have developed a lower bound, $lb2$, for GOLA instances. This lower bound is obtained by solving $(n-1)$ max flow problems. We begin by arbitrarily picking a pair (i, j) of vertices. The weight $w(i, j)$ of an edge is its capacity. The min cut (or equivalently, max flow) between these two vertices is computed. Let the value of this be v. The min cut partitions the vertices in V. One partition, A, includes i and the other, B, includes j (Figure 3.1(a)). The partitioning structure may be represented as in Figure 3.1(b). This partitioning process is now repeated on A (B) by keeping together all nodes in B (A) as one node. As a result of this, A and B are further partitioned to obtain the structure of Figure 3.1(c). The partitioning process is continued until each partition contains exactly one vertex. At this time, the partition structure will be an n-vertex tree. $lb2$ is the sum of the costs of the $(n-1)$ min cuts (or max flows) used to obtain this tree. When this tree is a chain, $lb2$ is the true minimum cost ([ADOL73]).

Since $lb2$ requires the computation of $(n-1)$ max flows, it is expensive to compute. Its suitability as a lower bounding function for a branch-and-bound algorithm is questionable. An easier to compute, and often more effective, lower bound, $lb3$, is obtained by ordering all the edges, $\{S_j \mid j \in C\}$, incident to a vertex, i, in nonincreasing order of weight. The weight of the first two edges is multiplied by 1, of the next two by 2, of the next two by 3, etc. The multiplied weights are summed. Let s_i be this sum. Then, $lb3 = \sum_{i \in V-P} s_i / 2$, where P is the set of vertices that have already been placed (we need to divide by 2 as each edge is incident to two vertices). The use of the multipliers 1, 1, 2, 2, 3, 3, ... is justified on the grounds that for any vertex i, at most two vertices can be adjacent to it in a linear arrangement; at most two can be distance 2 from it; etc.

The strategy of $lb3$ may be refined by taking into account the fact that if a vertex at position 1 or n of the linear arrangement has p adjacent vertices, then these must be at distances 1, 2, ..., p (and not 1, 1, 2, 2, 3, 3, ... as assumed in $lb3$). So, the multipliers 1, 2, 3, ..., p are to be used in computing s_i. If the vertex is at position 2 or $(n-1)$, then the multipliers 1, 1, 2, 3, ..., $p-1$ may be used, and so on. We construct a bipartite graph in which one set of vertices V represents the vertices in the GOLA instance and the other set Q represents the n positions of the linear arrangement. The cost of an edge (i, j) with $i \in V$ and $j \in Q$ is the value of s_i obtained as above under the assumption that vertex i is assigned to position j. A minimum assignment problem (between V and Q) is now solved ([PAPA82]). The cost of this minimum assignment is a lower bound, $lb4$, on the cost of C.

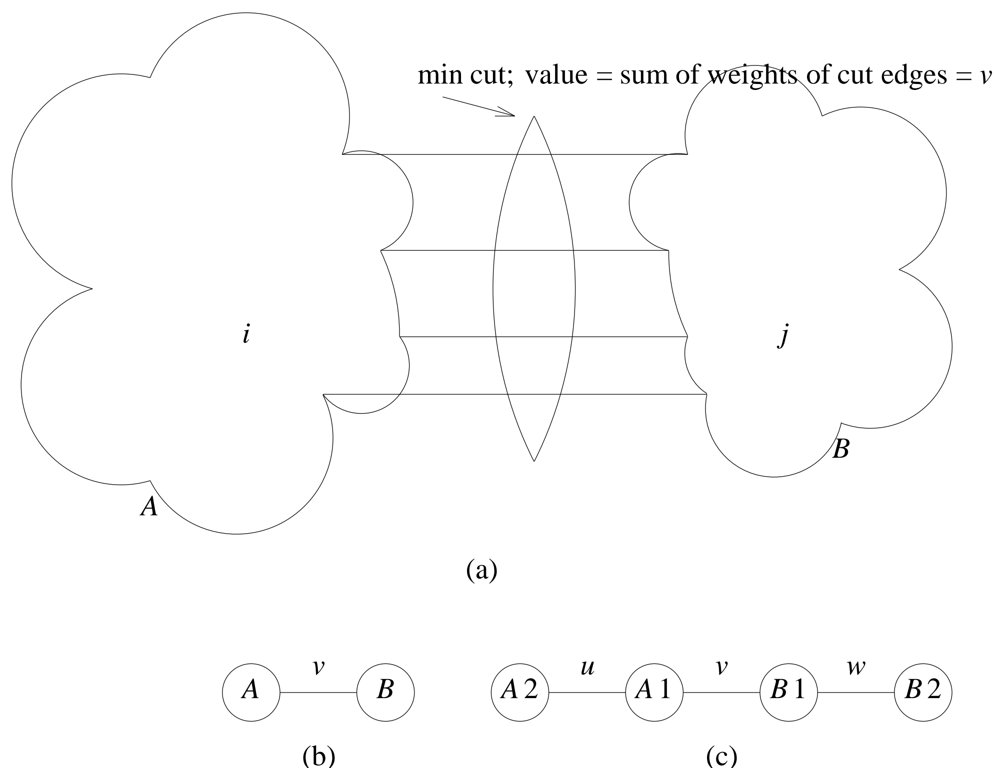

Figure 3.1: Min cut partitioning of graph to obtain an n-vertex tree.

3.1.2 Comparison of Lower bounds for C ... GOLA

The cost of computing the lower bounds $lb1$, $lb2$, $lb3$, and $lb4$, using classical algorithms for max flows and minimum assignment are, respectively, O(e), O(n^4), O($e\log e$), and O(n^3), where $e = |C|$ and $n = |\bigcup_{i \in C} S_i|$. For repeated computation of $lb3$, the edges may be sorted once by weight. The added cost of computing $lb3$ is O(e) per computation.

It is easily verified that $lb3$ provides a better bound than does $lb2$ for the following graphs: trees, complete graphs, cycles, star graphs, wheel graphs, and regular graphs.

$lb3$ is recommended over $lb2$ because it is simpler to compute and because it will generally give a better bound. $lb3$ will give a better bound than $lb1$. However, $lb4$ is expected to give the best bound. Since it is more expensive to compute, we do not recommend its use over $lb3$.

3.1.3 Lower bounds for *C* ... HOLA

$lb\,1$ and $lb\,2$ may be extended to the case of hypergraphs. $lb\,1$ simply takes the form, $\sum_{i \in C} w_i\,|S_i - 1|$. To extend $lb\,2$, the hypergraph under consideration is transformed into a graph by replacing each set S_i, $i \in C$, $p = |S_i|$, by $p\,(p-1)\,/\,2$ edges connecting the possible $p\,(p-1)\,/\,2$ pairs of vertices in S_i. The weight assigned to each of these edges is $w_i\,/\,(p-1)$. We see no easy way to adapt $lb\,3$ and $lb\,4$ to the case of hypergraphs.

Given the cost of computing $lb\,2$, we suspect better performance will be obtained using $lb\,1$.

3.1.4 An Example

Figure 3.2(a) shows an example GOLA instance. The part of the permutation state space tree generated when a least cost branch-and-bound is used with the bounding function $lb\,3$ is shown in Figure 3.2(b). Numbers marked adjacent to the nodes denote their $L(P)$ values. Nodes marked *B* are nodes that get killed because of the bounding function. In this instance, only 10 of the possible 64 nodes of the state space tree are generated. The optimal ordering is 1 - 2 - 3 - 4 and its cost is 9.

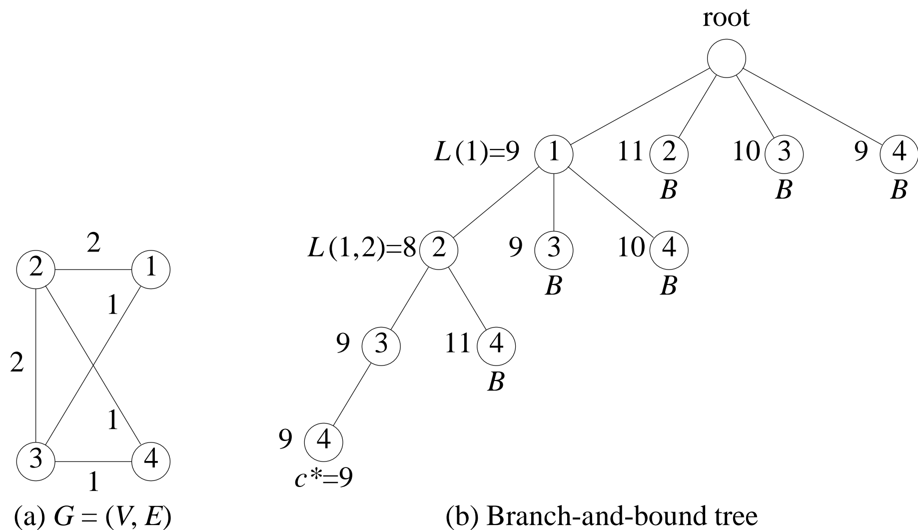

Figure 3.2: Example of branch-and-bound.

3.2 Dynamic Programming

The size of the solution space to be searched can be reduced from $n!$ to 2^n using dynamic programming. Consider any permutation $\sigma(V)$ of the vertex set. Divide this at any arbitrary point into a left and a right segment (Figure 3.3). Let P denote the vertices in the left segment and let $Q = V - P$ denote those in the right segment. Partition S with respect to P into three portions A, B, and C as in §3.1. Let $\gamma(P)$ be the ordering of P in $\sigma(V)$. Define *LeftCost* $(\gamma(P))$ as :

$$LeftCost\,(\gamma(P)) = L^A(P) + M^B(P),$$

where $L^A(P)$ is as in §3.1 and $M^B(P) = \sum_{i \in B} w_i\,(|P| - f_i)$, where f_i is as in §3.1. Next, partition S with respect to Q into A, B, and C (as in §3.1 but using Q in place of P). Let $\delta(Q)$ be the ordering of Q in $\sigma(V)$. Define *RightCost* $(\delta(Q))$ as:

$$RightCost\,(\delta(Q)) = L^A(Q) + N^B(Q),$$

where $L^A(Q)$ is as in §3.1 and $N^B(Q) = \sum_{i \in B} w_i\, l_i$; l_i is the position in $\delta(Q)$ of the rightmost vertex in S_i (the first vertex in $\delta(Q)$ is at position 1). Clearly,

$$cost\,(\sigma) = LeftCost\,(\sigma(V)) = RightCost\,(\sigma(V)) = LeftCost\,(\gamma(P)) + RightCost\,(\delta(Q)).$$

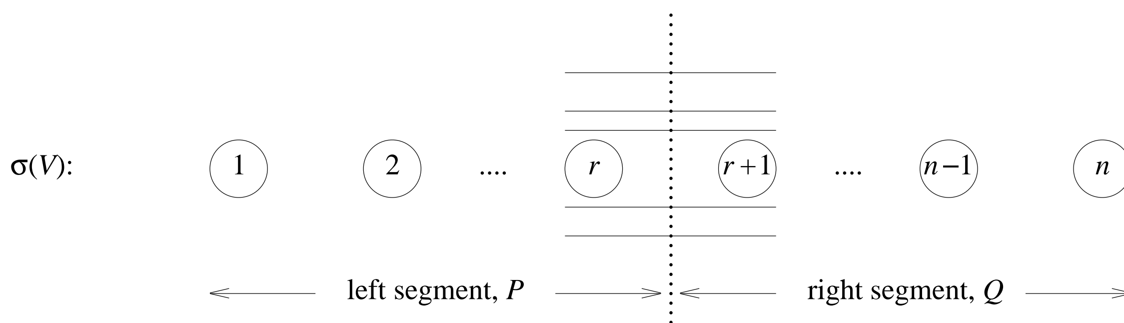

Figure 3.3: Partition of a permutation, $\sigma(V)$.

Hence, a minimum cost arrangement for V consists of a minimum cost arrangement for P and a minimum cost arrangement for Q. This means that we need not consider suboptimal arrangements for P and Q. Let *MinLeftCost* (P) be min $\{$ *LeftCost* $(\sigma(P) \mid \sigma$ is a permutation of $P)\}$. The correctness of the following recurrence for *MinLeftCost* follows from the preceding discussion:

$$MinLeftCost\ (P) = \begin{cases} 0, \text{ if } |P| \le 1 \\ \min_{j \in P}\{LeftCost\ (Perm\ (P,\ j))\}, \text{ if } |P| > 1 \end{cases}$$

where *Perm* (P, j) is the permutation obtained by adding j to the right end of the permutation of $P - \{j\}$ that has a *LeftCost* equal to *MinLeftCost* ($P - \{j\}$).

This recurrence may be solved using standard dynamic programming techniques (see [HORO78], for e.g.). We simply compute *MinLeftCost* (P) for all subsets P of V in the order, $|P| = 1, 2, 3, ..., n$. With each *MinLeftCost* (P), the permutation that has least cost is retained. The total number of *MinLeftCost* (P) values to be computed is the number of subsets of V (i.e., 2^n). If a trie (see [HORO84], for e.g.) is used to keep track of the various subsets of V, their *MinLeftCost*, and the associated permutations, then the dynamic programming algorithm just described will have a run time of $O(n(n+k)2^n)$ (recall that $n = |V|$ and $k = |S|$).

3.2.1 An Example

We apply the dynamic programming algorithm to the GOLA instance used in §3.1.4 (Figure 3.2(a)). Table 3.1 shows the results. *MinLeftCost* (P) for all subsets P of V is computed in the order of $|P| = 1, 2, 3, 4$. The permutation obtained by computing *MinLeftCost* ({1,2,3,4}) gives us the optimal ordering.

To see how the dynamic programming algorithm works on the given instance, let us try to compute *MinLeftCost* ({1,2,3}). At this point, we must have completed computing *MinLeftCost* for all subsets P of size 2. Using the above recurrence, we get *MinLeftCost* ({1,2,3}) = min {*LeftCost* (*Perm* ({1,2,3}, 1)), *LeftCost* (*Perm* ({1,2,3}, 2)), *LeftCost* (*Perm* ({1,2,3}, 3))}. *LeftCost* (*Perm* ({1,2,3}, 1)) = *LeftCost* (permutation from *MinLeftCost* ({2,3}), with {1} added at the right end) = *LeftCost* (3-2-1) = 9. Similarly, *LeftCost* (*Perm* ({1,2,3}, 2)) = *LeftCost* (1-3-2) = 8; *LeftCost* (*Perm* ({1,2,3}, 3)) = *LeftCost* (1-2-3) = 7. Thus, *MinLeftCost* ({1,2,3}) = 7 with the permutation of 1-2-3. To take another example, let us compute *MinLeftCost* ({1,2,3,4}). At this stage, *MinLeftCost* for all subsets of P of size 3 have been determined. Again using the recurrence, we get *MinLeftCost* ({1,2,3,4}) = min {*LeftCost* (*Perm* ({1,2,3,4}, 1)), *LeftCost* (*Perm* ({1,2,3,4}, 2)), *LeftCost* (*Perm* ({1,2,3,4}, 3)), *LeftCost* (*Perm* ({1,2,3,4}, 4))}. *LeftCost* (*Perm* ({1,2,3,4}, 1)) = *LeftCost* (permutation from *MinLeftCost* ({2,3,4}), with {1} added at the right end) = *LeftCost* (4-3-2-1) = 9. Similarly, *LeftCost* (*Perm* ({1,2,3,4}, 2)) = *LeftCost* (4-3-1-2) = 11; *LeftCost* (1-2-3-4) = 9; *LeftCost* (4-2-1-3) = 11. Thus, the permutation with the least cost of *MinLeftCost* ({1,2,3,4}) is 4-3-2-1, which gives us the optimal solution.

$\|P\| = 1$				
P	{1}	{2}	{3}	{4}
MinLeftCost	0	0	0	0
Permutation	1	2	3	4
$\|P\| = 2$				
P	{1,2}	{1,3}	{1,4}	
MinLeftCost	3	3	2	
Permutation	1-2	1-3	4-1	
P	{2,3}	{2,4}	{3,4}	
MinLeftCost	4	2	2	
Permutation	3-2	4-2	4-3	
$\|P\| = 3$				
P	{1,2,3}	{1,2,4}	{1,3,4}	{2,3,4}
MinLeftCost	7	7	6	6
Permutation	1-2-3	4-2-1	4-3-1	4-3-2
$\|P\| = 4$				
P	{1,2,3,4}			
MinLeftCost	9			
Permutation	4-3-2-1			

Table 3.1: Example of dynamic programming.

4 HEURISTICS

Since the approximation version of the HOLA problem is NP-hard, it is not surprising that the heuristics in use do not guarantee to find solutions that are a bounded distance (in value) from optimal. However, these heuristics do perform well in practice. Two different heuristic approaches are manifested in the work of Schuler and Ulrich [SCHU72] and Kang [KANG83]. Schuler and Ulrich adopt a clustering approach while Kang develops a heuristic based on the

greedy method. While Kang´s heuristic is stated for the case of unit weights ($w_i = 1$), it is easily extended to the case of arbitrary weights. Kang´s heuristic begins with a lightly connected vertex at the leftmost position and then builds the linear arrangement left to right adding one vertex at each iteration.

In this section, we develop an enhanced clustering heuristic and also propose two heuristics based on the Monte Carlo method. Our enhanced clustering heuristic performed at least as well as the heuristics of [SCHU72] and [KANG83] on all instances that we tested. Improved solutions are, however, obtained using the Monte Carlo methods.

4.1 Clustering

Our clustering heuristic, like that of [SCHU72], has two phases to it. The first generates a cluster tree and the second iteratively improves this tree. The clustering algorithm begins with n clusters. Each vertex of the hypergraph, H, is in a distinct cluster. In each iteration of the clustering algorithm, a pair of clusters are combined together to form a single cluster. This pair is chosen by computing the *attraction factor*, AF, for each pair of clusters. This is defined as:

$$AF(i,j) = \sum_{p=1}^{k} w_p \, \delta_p$$

where w_p = weight of subset S_p; $\delta_p = 1$ if both clusters i and j have a vertex in S_p, 0 otherwise. The pair of clusters (i, j) with the maximum attraction factor is combined to form a new cluster w. Ties are broken using the following rules:

Rule 1: Let W_i be the set of vertices in the cluster i. Let T_i be the set of subsets that have vertices in W_i, i.e., $T_i = \{ p \mid j \in W_i \text{ and } j \in S_p \}$. A set S_p is said to be *complete* in the cluster i iff $S_p \subseteq W_i$. Pairs (i, j), such that all subsets in T_i or all in T_j are complete in the cluster obtained by pairing i and j are given priority over other pairs.

Rule 2: Select a pair in which the largest number of sets complete.

Rule 3: Arbitrarily select a pair.

Rule 1 is applied first. If ties remain, then rule 2 is applied on the tied pairs. Any remaining ties are broken arbitrarily.

The combining of two clusters into one can be represented graphically as in Figure 4.1. Node i (j) is the left (right) child of node w. The choice between which of i and j should be the left child is arbitrary at this point.

The pairwise combination of clusters is continued until only one cluster remains. The sequence in which clusters are combined describes a binary tree in which the ordering between the left and right children is chosen arbitrarily. The clustering algorithm just described can be implemented to run in $O(n^2 k + n^2 \log n)$ time , where $n = |V|$ and $k = |S|$ (the implementation is

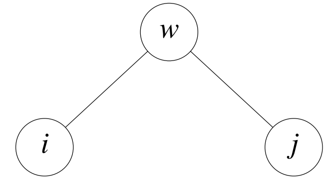

Figure 4.1: Cluster *w* formed by combining two clusters, *i* and *j*.

quite straightforward and employs balanced search trees (see [HORO84], for e.g.) to select minimum cost pairs). Example 4.1 illustrates the clustering process and the resulting tree. The left to right order of the leaf nodes defines a possible solution to the HOLA problem.

Example 4.1: Consider a 6-vertex instance with S = {{1,6}, {1,2,3}, {1,2}, {3,4,5}, {4,5}, {1,3,4}, {3,4}} and W = {4, 2, 1, 1, 3, 1, 1}. We start by placing each vertex in a cluster. The attraction factor for all pairs of clusters is determined. The cluster pairs with the maximum *AF* value of 4 are (1, 6) and (4, 5). Rule 1 is now applied to break the tie. Cluster pair (1, 6) succeeds since all its subsets in T_6 are complete. A new cluster, *a*, is formed by combining (1, 6). See Figure 4.2. *AF* values are computed for cluster pairs involving *a*. Cluster pair (4, 5) has the maximum *AF* value of 4. (4, 5) is combined to form a new cluster *b*. After forming new cluster pairs and computing their *AF* values, we determine (2, *a*), (3, *a*), and (3, *b*) to have the highest *AF* value of 3. Tie breaking rules are again applied. Rule 2 determines that (3, *b*) is to be selected over (2, *a*) and (3, *a*) because it has 2 complete sets compared to 1 and 0 for the latter two clusters, respectively. At this point, we have three clusters, *a*, 2, and *c*, left. *AF*(*a*, 2) = 3. *AF*(2, *c*) = 2. *AF*(*a*, *c*) = 3. The number of complete sets for both (*a*, 2) and (*a*, *c*) is 1. Rule 3 is applied and cluster pair (*a*, 2) is picked arbitrarily. The last two clusters, *c* and *d*, are combined to form a new cluster, *e*, which becomes the root of the cluster tree. The complete cluster tree is shown in Figure 4.2. □

Our clustering procedure is quite similar to that described in [SCHU72]. However, we use a different *attraction factor* function. Our function is more natural for the case when some (or all) of the S_i´s have more than two vertices. The next phase in the clustering algorithm is the improvement phase. Here, we attempt to reorder the children of nodes in the cluster tree so as to obtain a better linear ordering of the leaf nodes. Central to our improvement algorithm is a procedure that starts with a cluster tree and finds an ordering of the leaves that is optimal with

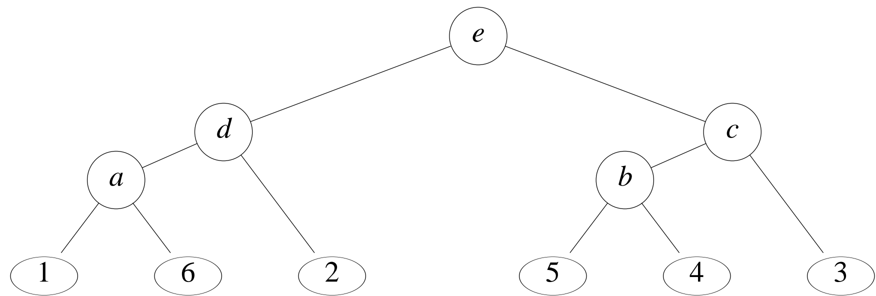

Figure 4.2: Cluster tree.

respect to a single inter exchange of the left and right subtrees of a node. This algorithm, procedure *BestTree*, is given in Figure 4.3.

Example 4.2: We apply procedure *BestTree* to the cluster tree, *T*, of Figure 4.2. The 6-vertex instance is defined in Example 4.1. Initially, node *e* is in the queue. This cluster is removed from the queue and its two children, *c* and *d*, are added to the end of the queue as they are non leaf nodes. The cost of the permutation with nodes *c* and *d* in their current position is 27. Note that the left to right order of the leaf nodes defines the ordering. The *c* and *d* subtrees are temporarily exchanged to give a new order 5 - 4 - 3 - 1 - 6 - 2. This has a cost of 20. This new configuration is accepted as it leads to a lower cost. The new cluster tree is shown in Figure 4.4(a). Node *d* is extracted from the queue and its non leaf child, *a*, is put at the end of the queue. The subtrees of node *d* are now exchanged giving a new permutation, 5 - 4 - 3 - 2 - 1 - 6. The cost of this is 18. Since this cost is again lower, the exchange of subtrees is accepted, giving a new cluster tree as shown in Figure 4.4(b).

Continuing, node *c* is removed from the front of the queue and its non leaf child, *b*, is added to the end of the queue. Nodes *b* and 3 are exchanged to give a new permutation of 3 - 5 - 4 - 2 - 1 - 6. The cost of this is 24. Since the cost is higher than the previous best cost, the exchange of the subtrees of node *c* is not accepted. Node *a* is deleted from the queue. Since both its children are leaves, no new nodes get added to the queue. The cost of the permutation on exchanging 1 and 6 is 22. Since this cost is higher than the best cost of 18, this exchange is also not accepted. The last node *b* is deleted from the queue and no new nodes get added since both of *b*´s children are leaves. Exchanging nodes 5 and 4 and computing the cost of the new ordering gives a value of 20. This exchange is also not accepted. At this point, we have completed one iteration of the

```
PROCEDURE BestTree (T);
    (* Traverse the cluster tree, T, in a breadth first manner. Exchange the left and right
    subtrees of a node if this exchange improves the ordering of the leaves. Iterate this
    traversal until no further improvement is possible *)
    REPEAT (* improvement loop *)
        • place the root into a queue;
        WHILE queue not empty DO (* breadth first traversal *)
            • extract node from queue front;
            • add the non leaf subtrees of this node to the queue end;
            • exchange the left and right subtrees of this node if this exchange
            reduces the cost of the leaf ordering;
        ENDWHILE;
    UNTIL no improvement in leaf ordering is made;
END BestTree;
```

Figure 4.3

repeat loop (procedure *BestTree*) and the best permutation is the left to right order of the leaf nodes in the cluster tree of Figure 4.4(b). On the second iteration of the repeat loop, no improvement in cost is found. Thus, the best order determined for our example hypergraph is 5 - 4 - 3 - 2 - 1 - 6, with a cost of 18. This also happens to be the optimal value. □

The complexity of *BestTree* is easily seen to be $O(r_1 n^2 k)$, where r_1 is the number of times the repeat loop is iterated. Experimental results provided later indicate that r_1 is about 2.

Our improvement algorithm is described in Figure 4.5. Procedure *Improve* essentially makes some transformations that are not possible in procedure *BestTree*. Procedure *Improve* examines the non leaf vertices of the cluster tree bottom up (top down examination is another possibility). This is done by first performing a breadth first traversal of the tree. During this traversal, all non leaf nodes are placed on a stack (Step 3). Because of the last-in-first-out property of the stack, Steps 4 and 5 examine the non leaf vertices in the desired bottom up order.

Let *u* be the vertex being examined. There are only three possibilities for its children:

1. Both are leaves: The orderings *ba* and *ab* (cf. Figure 4.6(a)) were tried by *BestTree* on its most recent invokation. So, nothing is to be gained by interchanging them.

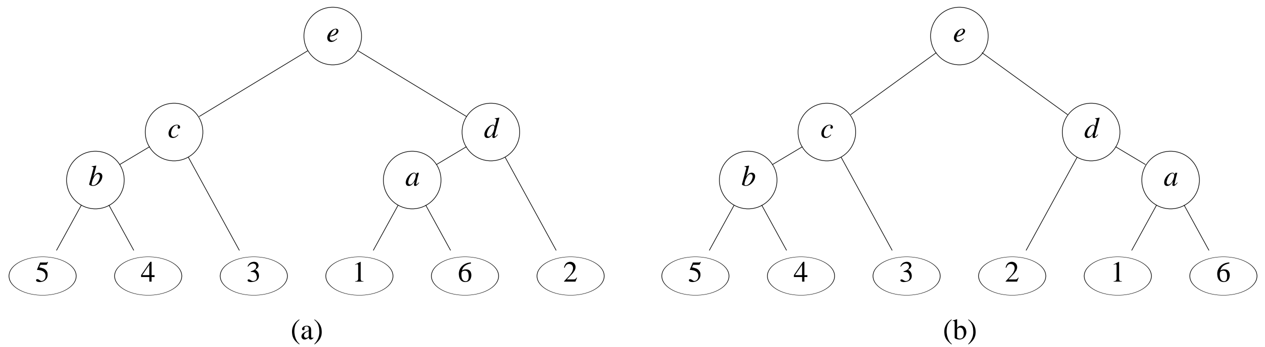

Figure 4.4: Tree configurations obtained during *BestTree* processing.

PROCEDURE *Improve* (*T*);

(* Obtain an improved ordering of the leaves of the cluster tree, *T* *)

Step 1: • Use *BestTree* (Figure 4.3) on *T*. Let the cost of the resulting ordering in the tree, *T*, be *BestCost*.

Step 2: REPEAT (* until no further improvement is possible *)

Step 3: • Traverse *T* in a breadth first manner storing all non leaf nodes on a stack;

Step 4: • Extract nodes from the top of this stack one at a time; FOR each node, *u*, extracted, DO Step 5;

Figure 4.5 (Continued on next page)

2. Exactly one child is a leaf: If the configuration is as in Figure 4.6(b), then the last invokation of *BestTree* tried the orderings {*acd*, *adc*, *cda*}. Restructuring *u* as in Figure 4.6(d) and invoking *BestTree* results in the orderings {*acd*, *cad*, *dac*} being tried. Similarly, transforming Figure 4.6(c) into Figure 4.6(e) results in orderings {*cda*, *dca*, *acd*, *cad*, *dac*} being tried.

Step 5: CASE

: both children of *u* are leaves:

• The situation is as in Figure 4.6(a). An exchange of the children was tried when *BestTree* was last invoked. So, do nothing now.

: exactly one child is a leaf:

• The situation is as in Figure 4.6(b) or (c). The subtree *u* in *T* is transformed to that of Figure 4.6(d) or (e), respectively, to get a new tree *T´*, and *BestTree* (*T´*) computed. If the resulting tree is of lesser cost than *BestCost*, then *BestCost* and *T* are updated.

: else: (* *u* has no leaf children *)

• At this time, the configuration is as in Figure 4.6(f). The subtree *u* of *T* is transformed to that of Figure 4.6(g) to get *T´*. *BestTree* is applied to *T´*. If the resulting tree is of lesser cost, then *T* and *BestCost* are updated. Next, subtree *u* of *T* of Figure 4.6(f) is transformed to that of Figure 4.6(h) to get *T´*. *BestTree* is applied to *T´* and if the resulting tree is of lesser cost, then *T* and *BestCost* are updated.

ENDCASE;

Step 6: UNTIL no improvement made; (* end of repeat of Step 2 *)

END *Improve*;

Figure 4.5 (Contd.)

3. Both children are non leaves: In the most recent invokation of *BestTree*, the orderings {*cdef*, *dcef*, *cdfe*, *efcd*} of *cdef* (cf. Figure 4.6(f)) were tried. The two transformations of Figures 4.6(g) and (h) result in the permutations {*cedf*, *ecdf*, *cefd*, *dfce*} and {*cfde*, *cfed*, *fcde*, *decf*} being tried, respectively. Note that the case when one child of *u* is null is not possible as *u* is a subtree of a cluster tree.

The complexity of *Improve* is $O(r_2 n^3 k)$, where r_2 is the number of iterations of the repeat loop of Steps 2 - 6. This analysis assumes that the number of iterations, r_1, of the repeat loop of *BestTree* is bounded by some constant. In practice, limiting r_1 to 2 is adequate.

The improvement phase of the heuristic of [SCHU72] is a single pass top to bottom

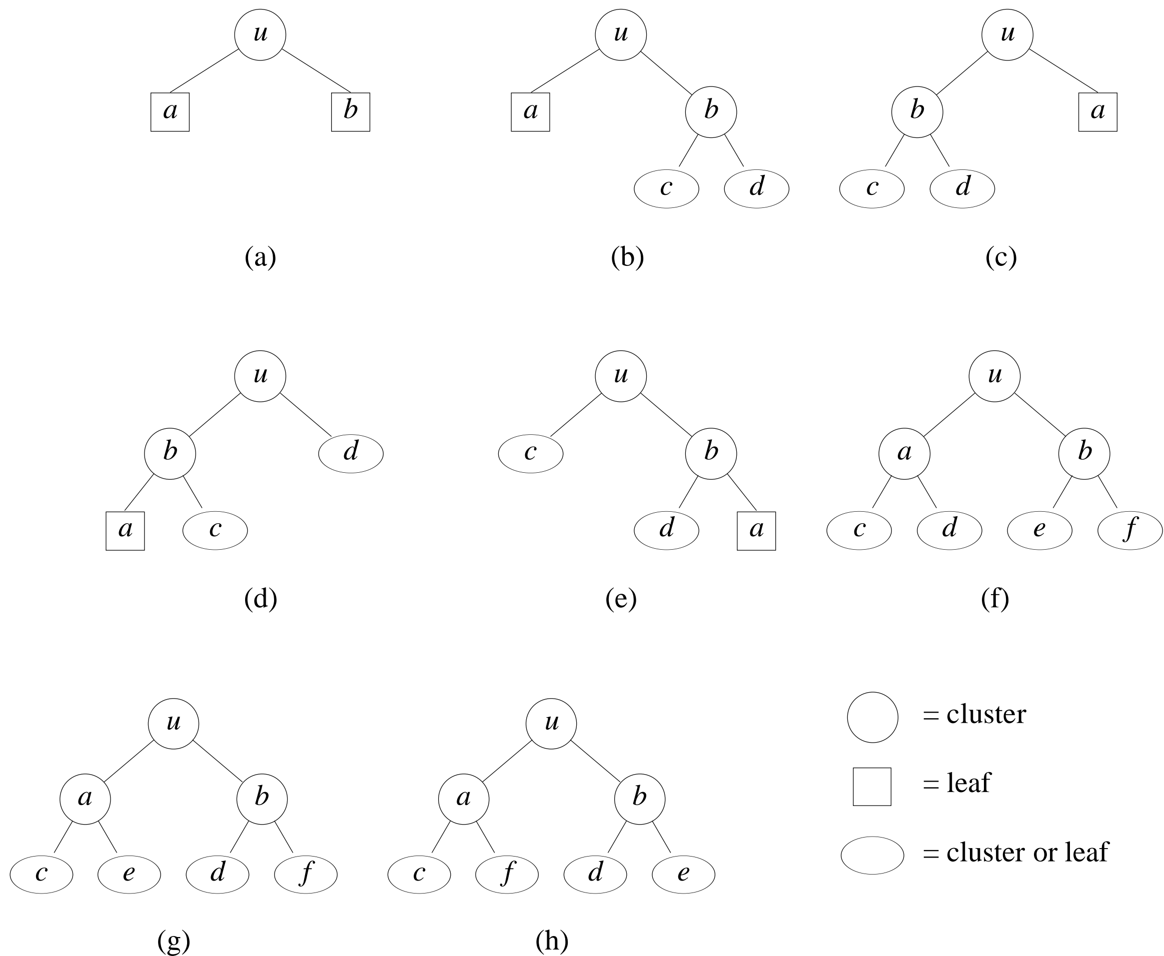

Figure 4.6: Various possibilities for subtree with root node *u* in procedure *Improve*.

algorithm. It corresponds to the first iteration of procedure *BestTree* and the cost of an ordering is computed in a modified manner.

4.2 Monte Carlo Methods

Nahar, Sahni, and Shragowitz [NAHA85] have experimented extensively with Monte Carlo methods (including simulated annealing) and recommend the two schemes, Strategies 1 and 2, shown in Figures 4.7 and 4.8, respectively. These differ essentially in that Strategy-2 permits a bad perturbation to be made only from a local optima while Strategy-1 permits such a

perturbation if a sufficiently long sequence of perturbations does not result in a better permutation.

Step 1: • Let σ_i be a random feasible solution to the given problem; *temp* = 1; *counter* = 0;

Step 2: • Let σ_j be a feasible solution that is obtained from σ_i as a result of a perturbation;

Step 3: IF $cost(\sigma_j) - cost(\sigma_i) < 0$ THEN

Step 4: [$\sigma_i = \sigma_j$; update best solution obtained so far in case σ_i is best;
counter = 0]

ELSE (* $cost(\sigma_j) - cost(\sigma_i) \geq 0$ *)

Step 5: [*counter* = *counter* + 1;

Step 6: IF *counter* > *maxcount* THEN

Step 7: [IF *temp* = *maxtemp* THEN [STOP];

Step 8: *temp* = *temp* + 1; *counter* = 0;

Step 9: • Retain the high cost solution σ_j as the starting solution for the next *temp*,
i.e., $\sigma_i = \sigma_j$]

ENDIF]

ENDIF;

Step 10: GO TO Step 2;

Figure 4.7: Monte Carlo method: Strategy-1.

We consider the two possible perturbation functions:

1. Single Insert: Move an element of the current permutation from its present position to another.
2. Pairwise Exchange: Exchange the positions of two elements of the current permutation.

4.3 Experimental Evaluation

The four Monte Carlo algorithms, the clustering algorithm, and the algorithm of [KANG83] were programmed in Pascal and run on an Apollo DN320 workstation. We did not program the clustering algorithm of [SCHU72] as it was clear to us that our clustering algorithm would generate better solutions and require more computer time.

Step 1: • Let σ_i be a random feasible solution to the given problem; *counter* = 0;

Step 2: • Continue to perturb σ_i until no perturbation results in a decrease in *cost*;

Step 3: IF σ_i is best THEN

Step 4: [Update the best solution found so far; *counter* = 0]

ELSE

Step 5: [*counter* = *counter* + 1]

ENDIF;

Step 6: IF *counter* > *maxcount* THEN [STOP];

Step 7: • Let σ_j be the result of a perturbation to σ_i; $\sigma_i = \sigma_j$;

Step 8: GO TO Step 2;

Figure 4.8: Monte Carlo method: Strategy-2.

Our initial experiment included 24 randomly generated instances with *n* ranging from 10 to 100, *k* ranging from 5 to 80, and having an average set size of 6. 15 of these instances were with unit set weights and the remaining had set weights ranging from 1 to 5. The conclusions from this experiment were:

1. The invokation of *BestTree* from Step 1 of procedure *Improve* resulted in no more than 2 iterations of the repeat loop of *BestTree*. I.e., $r_1 \approx 2$.
2. The number of iterations of the repeat loop of procedure *Improve* is between 5 and 7. I.e., $r_2 \approx 7$.
3. The ordering obtained after Step 1 of *Improve* is comparable to that obtained by Kang´s heuristic. Actually, it was better than Kang´s solution on 10 of the 24 instances; it was the same for 2 instances; and worse on the remaining. However, Kang´s algorithm is much faster.
4. The ordering obtained by *Improve* was never inferior to that obtained by [KANG83]. But the run time of *Improve* is much larger. Table 4.1 gives the results for 12 of the 24 instances. STEP1 is the permutation obtained following Step 1 of procedure *Improve*.
5. On the 31-vertex, 33 subsets (nets) example of [SCHU72], the heuristic of [KANG83] produces a solution that has cost 95. The solution produced by [SCHU72] has cost 98. The

Instance #	[KANG83]		STEP1		*Improve*	
	cost	time	cost	time	cost	time
1	705	0.7	705	5.7	671	45.9
2	1375	1.2	1472	24.2	1348	200.0
3	187	0.4	177	1.0	170	8.6
4	452	0.8	382	4.7	326	43.0
5	720	0.6	707	5.4	650	36.4
6	564	1.2	534	23.7	489	134.0
7	121	1.0	113	5.8	105	66.7
8	528	0.8	537	11.6	523	95.8
9	1904	1.1	1928	24.3	1702	180.0
10	788	1.7	795	50.9	705	364.0
11	316	1.6	296	23.0	252	217.0
12	37	0.5	40	1.0	32	6.1

*(time is in seconds)

Table 4.1

solution produced by *Improve* has cost 91. The solution of [KANG83] can however be improved to one with cost 89 using Strategy-2 of the Monte Carlo methods and the pairwise exchange perturbation method. The linear ordering obtained is: OP1 - U - OP3 - V - W - X - OP2 - Y - Z - P2 - P - T - S - O - N - K - L - M - Q - R - D - E - F - I - J - H - C - G - A - B - P1 (see [SCHU72] for the instance characteristics).

6. Of the four Monte Carlo methods (with *maxcount* = 18, *maxtemp* = 6), the two single exchange methods did not produce as good solutions as obtained by procedure *Improve*. These methods did, however, require less computing time than when pairwise exchange is used.

7. Of the two pairwise exchange methods, pairwise exchange with Strategy-2 consistently outperformed pairwise exchange with Strategy-1. The former also took more time. However, providing more time to the latter by increasing *maxcount* to 25 did not result in better

solutions. Tables 4.2 and 4.3 show the results obtained for the first six instances used for Table 4.1. Three different starting permutations were used for each instance: [KANG83] is the permutation obtained from the greedy heuristic of [KANG83], STEP1 is the permutation obtained following Step 1 of procedure *Improve*, and RANDOM is a randomly generated permutation. "Start" denotes the cost of the randomly generated permutation. We did not attempt to use the permutation produced by procedure *Improve* as the run time of this procedure was considered too high to justify its use to obtain an initial permutation.

Instance #	[KANG83]		STEP1		RANDOM		
	Str-1	Str-2	Str-1	Str-2	Start	Str-1	Str-2
1	668	660	674	652	834	701	652
2	1368	1245	1400	1276	1925	1500	1243
3	172	160	177	160	251	177	160
4	384	301	382	303	778	417	303
5	720	643	684	643	1137	729	643
6	564	472	532	472	841	639	470

*(All entries denote cost)

*(Str-1 is Strategy-1 of Figure 4.7; Str-2 is Strategy-2 of Figure 4.8)

Table 4.2: Solution costs using pairwise exchange.

Because of the overall good performance of the pairwise exchange Monte Carlo methods, we decided to see if these could be further improved. We experimented with a modification of Figures 4.7 and 4.8 in which Step 9 of Figure 4.7 and Step 7 of Figure 4.8 was changed to:

- Let σ_i be obtained from the best permutation found so far by a single random pairwise exchange.

In all our tests, this change yielded better solutions. Our final experiment involved two sets of 30 random instances. The first set consisted of instances with $k = n$ while the instances of the second had $k = 4n$. In the first set, n ranged from 15 to 70 and in the second, from 15 to 40. On the average, a set had six vertices. Set weights ranged from 1 to

Instance #	[KANG83]		STEP1		RANDOM	
	Str-1	Str-2	Str-1	Str-2	Str-1	Str-2
1	1.6	48.0	1.3	75.0	1.4	6.8
2	0.8	76.0	2.7	20.0	2.8	350.0
3	1.2	18.4	0.5	2.2	0.5	5.6
4	1.3	39.0	1.6	25.0	1.6	12.9
5	0.6	30.0	0.7	16.0	1.2	3.1
6	0.4	211.0	0.4	142.0	2.7	118.0

*(All entries denote time in seconds)

Table 4.3: Computing time using pairwise exchange.

Row #	# of iterations	[KANG83]		STEP1		RANDOM	
		Str-1	Str-2	Str-1	Str-2	Str-1	Str-2
1	1	2	28	1	29	4	26
2	10	10	19	4	25	14	16
3	1	0	6	0	20	1	3
4	10	2	6	2	17	7	6

*(Entries in table denote # of instances)

*(Str-1 = Strategy-1; Str-2 = Strategy-2)

Table 4.4: Instances with $k = n$.

Row #	# of iterations	[KANG83]		STEP1		RANDOM	
		Str-1	Str-2	Str-1	Str-2	Str-1	Str-2
1	0	56134	56134	57022	57022	81738	81738
2	10	51470	50796	50945	50027	51233	51031
3	1	54716	53169	53556	51175	54560	52773
4	0→1	2.5	5.3	6.1	10.3	33.3	35.4
5	0→10	8.3	9.5	10.7	12.3	37.3	37.6
6	1→10	5.9	4.5	4.9	2.2	6.1	3.3

*(numbers in top part of table denote sum cost of all 30 instances)

*(numbers in second part of table denote percentage improvement)

Table 4.5: Instances with $k = n$.

5. The results are summarized in Tables 4.4 - 4.7. Tables 4.4 and 4.5 give the results for the thirty instances with $k = n$ while the results for $k = 4n$ are given in Tables 4.6 and 4.7. The modified Monte Carlo algorithms were used. Each instance was solved using the same three starting permutations as used before in Table 4.2. Rows 1 and 2 of Tables 4.4 and 4.6 give the number of instances on which one method performed better than another (i.e., pairwise exchange with Strategy-1 versus pairwise exchange with Strategy-2). Both methods were given the same amount of time. In the case of row 1, this time was adequate for Strategy-2 to make 1 iteration (Step 2 of Figure 4.8 is executed only once), while in the case of row 2, it was adequate for 10 iterations (Step 2 of Figure 4.8 is executed in the algorithm 10 times). From row 2 of Table 4.6, we see that using a random start and adequate time for 10 iterations, Strategy-2 outperformed Strategy-1, 19 times, Strategy-1 outperformed Strategy-2, 7 times. On the remaining 4 instances, both yielded equally good linear arrangements. Rows 3 and 4 compare solutions over all six columns. Thus from row 4 in Table 4.6, we see that using adequate time for 10 iterations, Strategy-2 obtained the best solution on 15 instances using a random starting arrangement, while the same strategy starting from STEP1 yielded the best solution on 14 instances. The total across rows 3 and 4 exceeds 30 as on some instances more than one method obtained the best solution.

Row #	# of iterations	[KANG83]		STEP1		RANDOM	
		Str-1	Str-2	Str-1	Str-2	Str-1	Str-2
1	1	4	21	3	26	7	22
2	10	9	18	6	21	7	19
3	1	4	8	2	15	1	3
4	10	5	10	3	14	5	15

*(Entries in table denote # of instances)

*(Str-1 = Strategy-1; Str-2 = Strategy-2)

Table 4.6: Instances with $k = 4n$.

Tables 4.5 and 4.7 give the improvements in the cost obtained by the Monte Carlo methods. Iteration 0 denotes the starting permutation. Since the starting permutation for both Strategy-1 and Strategy-2 are the same for a given type of start, their entries in the table are also identical (see row 1 in Tables 4.5 and 4.7). Since random initial solutions have significantly higher cost than those obtained from [KANG83] or STEP1, it is not surprising that the improvement obtained for these random solutions is far more significant.

From these tables, we see that the best solutions are obtained using the solution of STEP1 as a starting arrangement and then improving it using the improved version of the pairwise Monte Carlo method of Figure 4.8 (Strategy-2).

5 CONCLUSIONS

We have carried out a thorough investigation of the Hypergraph Optimal Linear Arrangement problem. Algorithms for optimal solutions, and for good solutions, as well as complexity results have been presented. A practical scheme to obtain reasonably good solutions is to use the modified pairwise exchange Monte Carlo method of Figure 4.8 beginning with the solution from Step 1 of procedure *Improve*. Beginning with a random solution yields almost as good results.

Row #	# of iterations	[KANG83]		STEP1		RANDOM	
		Str-1	Str-2	Str-1	Str-2	Str-1	Str-2
1	0	100227	100227	103179	103179	117952	117952
2	10	96802	96755	96917	96510	96871	96585
3	1	99332	98952	99128	97608	99344	98252
4	0→1	7.0	1.2	3.9	5.4	15.8	16.7
5	0→10	3.4	3.4	6.1	6.5	17.9	18.1
6	1→10	2.5	2.2	2.2	1.1	2.4	1.7

*(numbers in top part of table denote sum cost of all 30 instances)

*(numbers in second part of table denote percentage improvement)

Table 4.7: Instances with $k = 4n$.

6 REFERENCES

ADOL73 Adolphson D. and T.C.Hu, *Optimal Linear Ordering*, SIAM J. Appl. Math., 25, 1973, pp 403-423.

COHO83 Cohoon J. and S.Sahni, *Heuristics for the Board Permutation Problem*, Proc. 1983 IEEE ICCAD Conference.

EVEN75 Even S. and Y.Shiloach, *NP-Completeness of Several Arrangement Problems*, Technical Report #43, Computer Science Dept., The Technion, Haifa, Israel, 1975.

EVEN79 Even S., *Graph Algorithms*, Computer Science Press, Rockville, Maryland, 1979.

GARE76 Garey M.R., D.S.Johnson and L.Stockmeyer, *Some Simplified NP-Complete Graph Problems*, Theoretical Computer Science, 1, 1976, pp 237-267.

GARE79 Garey M.R. and D.S.Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H.Freeman, San Francisco, 1979.

GOMO61 Gomory R.E. and T.C.Hu, *Multi-Terminal Network Flows*, J. Soc. Indust.

Appl. Math., 9, 1961, pp 551-570.

HANA72 Hanan M. and J.M.Kurtzberg, *Placement Techniques*, Chapter 5, Design Automation of Digital Systems, Theory and Techniques, Vol 1, Prentice Hall, Englewood Cliffs, New Jersey, 1972.

HORO78 Horowitz E. and S.Sahni, *Fundamentals of Computer Algorithms*, Computer Science Press, Potomac, MD, 1978.

HORO84 Horowitz E. and S.Sahni, *Fundamentals of Data Structures in Pascal*, Computer Science Press, Rockville, MD, 1984.

KANG83 Kang S., *Linear Ordering and Application to Placement*, Proc. 20th DAC, 1983, pp 457-464.

NAHA85 Nahar S., S.Sahni and E.Shragowitz, *Experiments with Simulated Annealing*, Proc. 22nd DAC, 1985, pp 748-752.

PAPA82 Papadimitriou C.H. and K.Steiglitz, *Combinatorial Optimization: Algorithms and Complexity*, Prentice Hall, Inc., Englewood Cliffs, New Jersey, 1982.

SCHU72 Schuler D.M. and E.G.Ulrich, *Clustering and Linear Placement*, Proc. 9th DAC, 1972, pp 57-62.

SHIL79 Shiloach Y., *A Minimum Linear Arrangement Algorithm for Undirected Trees*, SIAM J. Computing, Vol 8, No 1, Feb 1979, pp 15-32.