Linear Placement Problem

Jing Lee

Department of Electronic Engineering

Southern Taiwan University of Technology

Tainan, Taiwan 701, R.O.C.

Email: leejing@mail.stut.edu.tw

The linear placement problem is to put the components of an electronic circuit into a linear sequence as to minimize certain cost function associated with each ordering. Typically two objectives are considered. The first is to minimize the total routing length and the second is to minimize the maximum cut value. This problem is known to be NP-hard, even if the circuit is modeled by a graph [Gar79]. No existing methods can guarantee optimum solution for large scale problems. Therefore, algorithms based on heuristic rationales are usually employed. In most practical implementations, constructive algorithms are used in the initial placement and the results are refined further by various iterative improvement algorithms. Since a good initial solution always quickly leads to a better final solution, it is desirable to develop a sound constructive algorithm to begin with [Han76].

Several strategies have been used for the constructive algorithms, such as **global methods** [Qui79, Che84], **branch and bound method** [Got77], **cluster growth methods** [Kan83, Sch72, Cox80], and **graph partitioning method** [Che87]. Theoretically, global methods seem to be more reasonable than others, because they consider all of the interconnections simultaneously and place all components in parallel to find a global solution. However, they have two main disadvantages. First, they use a complete graph to model the interconnection net topology. For multipin nets, there are a disproportionately large impact on the placement [Sch72]. Second, they use a quadratic objective function which heavily penalizes long nets. Thus routing lengths tend to cluster tightly around the mean net length. The branch and bound method obtains a suboptimal solution whose cost is within $(1 + \varepsilon)$ times of the optimal cost. However, the computation time rises exponentially as the value of ε decreases. Thus, it is used only in small-sized problems. Cluster growth algorithms are based on bottom-up technique. They are fast but produce poor results because incomplete information is used to make placement decisions. On the other hand, graph partitioning algorithms are based on a top-down strategy. The interconnection information at a global level is considered in the placement decisions. Therefore they produce better results than cluster growth algorithms. However, they are computationally more expensive [Mur80]. Furthermore, they are based on sequential optimization process, thus the minimization objective is only achieved locally.

While the initial placement strategies surveyed have some desirable characteristics, they also exhibit undesirable features as discussed above. In an attempt to overcome this

situation, a new algorithm which combines the efficiency of top-down heuristics with the speed of bottom-up approaches is presented. The objective is to minimize both the total routing length and the tracks required.

The remainder of this paper is organized as follows. The linear placement problem and associated glossaries are described in Section 4.1.2. In Section 4.1.3, a new efficient algorithm is presented. Space and time complexities of the algorithm are analyzed in Section 4.1.4. Experimental results and discussions are given in Section 4.1.5. Finally conclusions are made in Section 4.1.6.

4.1.1 **Formulation of Linear Placement Problem**

Given a hypergraph H = (V, E), the vertex set V contains m vertices indexed by $\{v_1, v_2, ..., v_m\}$, and the hyperedge set E = $\{e_1, e_2, ..., e_n\}$. Here, an atom is an unplaced chip ;

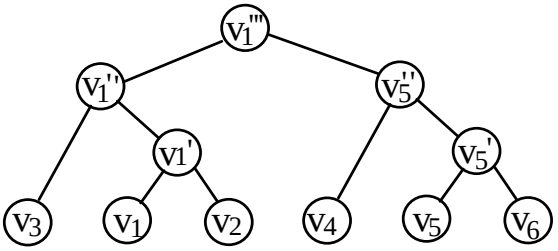


Fig. 4.1.1 A cluster tree. The ordering in v_1''' is $<v_3, v_1, v_2, v_4, v_5, v_6>$

a cluster is a linear set of placed chips. A cluster v_i of m constituent atoms, denoted by $<v_i^1, v_i^2, ..., v_i^k>$, indicates the ordering of the constituent vertices in v_i from left to right. It is implemented by a binary tree structure called cluster tree. Fig. 4.1.1 shows an example for the cluster tree, where the vertex, $v_1''' = <v_3, v_1, v_2, v_4, v_5, v_6>$.

A placement is represented by $<v^1, v^2, ..., v^m>$. The cut value between any two physically neighboring vertices v^i and v^{i+1} in a linear placement is represented by $C_V(v^i, v^{i+1})$. The maximum cut value is

$$\operatorname{Max}_{i=1}^{m-1} C_V(v^i, v^{i+1})$$

The width of a vertex is determined by its physical size. Without loss of generality, we assume that all vertices have the same width. Thus, the total routing length can be defined by

$$\sum_{i=1}^{m-1} C_V(v^i, v^{i+1})$$

The linear placement problem can be stated as follows :

> ***Given an electrical circuit consisting of fixed blocks, and a netlist interconnecting terminals on the periphery of these blocks and on the periphery of the circuit itself, construct a vertex ordering with minimized total routing length or with minimized maximum-cut-value.***

For many years, the objective of **minimum routing length** has been employed. However, this sometimes can lead to unroutable crowded regions, especially in the middle of a placement. On the other hand, the **min-cut** objective can avoid interconnections crowding in the placement and provides better routability [Wip82]. But it pays no attention to the wire length and can lead to excessively long nets. In order to avoid those undesirable features, several algorithms based on combined objectives have been proposed [Kan83, Xu87, Sch76]. The objective used in the paper, which will be defined in Section 4.1.2.1, also is a combination of the above two. It tends to produce a placement with uniform distribution of nets.

4.1.2 **Algorithm Description**

The present algorithm seeks a solution using heuristic rules in a sequential deterministic manner. It includes three phases : **SELECTION**, **PLACEMENT**, and **REDUCTION**. First, the SELECT function chooses a vertex-pair to be placed, one at a time, based on selection rules. Then, the PLACE function decides the relative positions for the selected vertex-pair. Finally, the REDUCE function reduces the current hypergraph to a shrunk hypergraph by combining the selected vertex-pair. The above process, called vertex combination process, is repeated until the hypergraph is reduced to one vertex. An outline of the algorithm is described in the following.

procedure Vertex_Combination_Algorithm
while |V| > 1 **do**
    SELECT a vertex-pair to be placed.
    PLACE the selected vertex-pair.
    REDUCE current hypergraph by combining the selected vertex-pair.
repeat Vertex_Combination_ALGORITHM
end Vertex_Combination_Algorithm

4.1.2.1 **SELECTION**

The SELECT function chooses a vertex-pair to be placed. For a hypergraph with m vertices and N hyperedges, there are $0.5\times m\times(m-1)$ vertex-pairs in the set of all possible vertex-pairs. It is time-consuming to select the best vertex-pair in the set, thus we reduce the set by the following two steps :

(1) For each hyperedge e_d, select the best vertex-pair, called candidate, in $V(e_d)$. All candidates construct a candidate set.

(2) Choose the best vertex-pair in the candidate set.

Through the two steps, the time complexity of selecting the best vertex-pair is only O(N).

Now, let us consider the vertex combination process as shown in Fig. 4.1.2(a) and (b). Here, vertex-pair v_i, v_j is selected, and then combined to form a cluster v_n. The set of hyperedges incident upon v_i or v_j is represented by $E(v_i, v_j)$, where $E(v_i, v_j) = E(v_i) \cup E(v_j)$. It can be divided into three disjoint subsets :

$E_{in}(v_i, v_j)$, $E_{pi}(v_i, v_j)$, and $E_{ex}(v_i, v_j)$. We call them the set of internal hyperedges, of external hyperedges, and of partial internal hyperedges, respectively. The definitions are :

$E_{in}(v_i, v_j) = \{e_d \mid V(e_d) = \{v_i, v_j\}\}$;
$E_{pi}(v_i, v_j) = \{e_d \mid V(e_d) \supset \{v_i, v_j\}\}$;
$E_{ex}(v_i, v_j) = \{e_d \mid v_i \in V(e_d) \vee v_j \in V(e_d), \{v_i, v_j\} \not\subset V(e_d)\}$.

An example shows in Figs. 4.1.2(a) and (b), where $E(v_i, v_j) = \{e_1, e_2, e_3, e_4\}$; $e_1 \in E_{in}(v_i, v_j)$; $e_2 \in E_{pi}(v_i, v_j)$; $\{e_3, e_4\} \in E_{ex}(v_i, v_j)$. After combining v_i and v_j to form the cluster v_n, the current hypergraph is reduced to a shrunk hypergraph as shown in Fig. 4.1.2(b). The pin number of v_n, $|E(v_n)|$ is equal to

$$|E(v_i)| + |E(v_j)| - 2\times|E_{in}(v_i, v_j)| - |E_{pi}(v_i, v_j)| \tag{4.1}$$

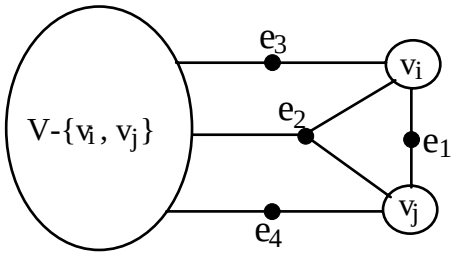
The difference in the number of pins between the current and the shrunk hypergraphs is

$$|E(v_n)| - |E(v_i)| - |E(v_j)| = -2\times|E_{in}(v_i, v_j)| - |E_{pi}(v_i, v_j)| \tag{4.2}$$

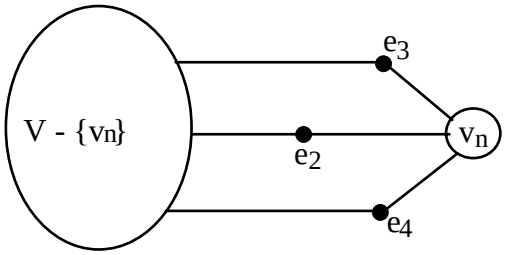
Since the total routing length of a hypergraph is related to its pin numbers, the above measure can be regarded as a measure of reducing the total routing lengths from the hypergraph after the combination process. It can be adopted as an evaluation function for an algorithm which is based on a sequential optimization process with the objective of minimizing the total routing lengths.

Another measure, $|E(v_n)|$, which is the cut value between v_n and its two topological neighbors, is equal to

$$|E_{ex}(v_i, v_j)| + |E_{pi}(v_i, v_j)| \tag{4.3}$$



(a) The previous hypergraph.



(b) Shunk hypergraph after combining v_i and v_j to form v_n.

Fig. 4.1.2 Vertex combination and the classify of hyperedges. $E(v_i, v_j) = \{e_1, e_2, e_3, e_4\}$; $e_1 \in E_{in}(v_i, v_j)$; $e_2 \in E_{pi}(v_i, v_j)$; $\{e_3, e_4\} \in E_{ex}(v_i, v_j)$.

Algorithms based on min-cut objective usually minimize the quantity.

A new objective function which is a combination of the above two measures, called IOC (inside-outside connectivity), is proposed as the objective function. It is defined by

$$\begin{aligned} IOC(v_i, v_j) &= (2) + 2\times(3) \\ &= 2\times\left|E_{ex}(v_i, v_j)\right| + \left|E_{pi}(v_i, v_j)\right| - 2\times\left|E_{in}(v_i, v_j)\right| \end{aligned} \tag{4.4}$$

The factor 2 accounts for that the internal and the external hyperedges are equally important.

The new objective function is more reasonable than measure (2) or (3) alone. It considers both the routing length and the cut value, thus it reflects the routability of signal nets more precisely. It should be pointed out that the new objective function is an improvement of a previous one [Kan83, Sch76] which is simply defined by

$$\left|E_{ex}(v_i, v_j)\right| - \left|E_{in}(v_i, v_j)\right| \tag{4.5}$$

Obviously, this objective function misses the contribution of partial internal hyperedges.

The rules of choosing the to-be-placed vertex-pair are

(1) Select the one with minimum IOC, if tie

(2) Select the one with minimum size sum, if tie again
(3) Select the one with maximum size difference.

The tie breaking scheme has been devised after numerous experiments.

4.1.2.2 **PLACEMENT**

The PLACE function decides the best relative ordering of the selected vertex-pair. The ordering is decided by a cost function which measures the routing length in the placement. Before describing the cost function, we have to define the topological coordinate of a hyperedge. Consider the hyperedge e_d which connects several constituent vertices in v_i as shown in Fig. 4.1.3, where $v_i = <v_i^1, ..., v_i^p>$. Let v_i^l and v_i^r be the leftmost and the rightmost constituent vertices attached to e_d, respectively. The topological coordinate of e_d

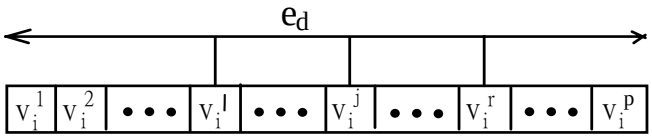


Fig. 4.1.3 The topological coordinate of e_d is (l, r)

in v_i is defined by (l, r), where l and r are the order number in v_i. In the study, the notations (l_i^d, r_i^d) are used to represent the topological coordinate of e_d in v_i.

Consider a selected vertex-pair v_i, v_j, where $v_i = <v_i^1, ..., v_i^p>$ and $v_j = <v_j^1, ..., v_j^q>$. The connection cost of the linear set $<v_i^1 ..., v_i^p, v_j^1, ..., v_j^q>$, is defined by

$$\sum_{e_d} \beta_d\left(\ell_j^d + m - r_i^d\right) + \mathrm{Min}\left(\sum_{e_d} \gamma_d \times \bar{\ell}^d, \sum_{e_d} \gamma_d \times \bar{r}^d\right) \tag{4.6}$$

where

$$\left(\bar{\ell}^d, \bar{r}^d\right) = \begin{cases} \left(\ell_i^d, n - r_j^d + 1\right) & \text{if} \quad \{v_i, v_j\} \subseteq V(e_d) \\ \left(\ell_i^d, m + n + 1 - r_i^d\right) & \text{if} \quad v_i \in V(e_d) \wedge v_j \notin V(e_d) \\ \left(m + \ell_j^d, n + 1 - r_j^d\right) & \text{if} \quad v_i \notin V(e_d) \wedge v_j \in V(e_d) \end{cases}$$

and

$$(\beta_d, \gamma_d) = \begin{cases} (1,0) & \text{if} \quad e_d \in E_{in}(v_i, v_j) \\ (1,1) & \text{if} \quad e_d \in E_{pi}(v_i, v_j) \\ (0,1) & \text{if} \quad e_d \in E_{ex}(v_i, v_j) \end{cases}$$

Seven ordering types must be considered, namely:

$$RR(v_i, v_j) = <v_i^1, ..., v_i^p, v_j^q, ..., v_j^1>$$
$$LL(v_i, v_j) = <v_i^p, ..., v_i^1, v_j^1, ..., v_j^q>$$
$$LR(v_i, v_j) = <v_i^p, ..., v_i^1, v_j^q, ..., v_j^1>$$
$$RL(v_i, v_j) = <v_i^1, ..., v_i^p, v_j^1, ..., v_j^q>$$
$$RR_r(v_i, v_j) = <v_j^1, ..., v_j^q, v_i^p, ..., v_i^1>$$
$$LL_r(v_i, v_j) = <v_j^q, ..., v_j^1, v_i^1, ..., v_i^p>$$
$$LR_r(v_i, v_j) = <v_j^1, ..., v_j^q, v_i^1, ..., v_i^p>$$

It can be noted that the last three are the reversion of the first three. If the selected vertices are not boundary vertices, only the first four need to be considered. The PLACE function compares the connection cost of different ordering types and then selects the one with minimum cost.

4.1.2.3 **REDUCTION**

Two basic operators in the REDUCE function : cluster tree construction and hypergraph reduction. In the cluster tree construction, two selected cluster trees vi and v_j are adjoined with root v_n. Before linking them together, the vertex orderings of the two cluster trees have to be rearranged according to the ordering type. For example, if $v_i = <v_i^1, ..., v_i^m>$ and $v_j = <v_j^1, ..., v_j^n>$ are adjoined with the ordering type of $LR(v_i, v_j)$, then both v_i and v_j have to reverse these orderings as $<v_i^m, ..., v_i^1>$ and $<v_j^n, ..., v_j^1>$, respectively. In the hypergraph reduction stage, the current hypergraph is reduced to a shrunk hypergraph by combining the selected vertex-pair $\{v_i, v_j\}$ to form a cluster v_n. The shrunk hypergraph is constructed by the following steps.

(1) $V \leftarrow V - \{v_i, v_j\}+\{v_n\}$.

(2) $E \leftarrow E - E_{in}(v_i, v_j)$.

(3) For $e_d \in E_{pi}(v_i, v_j) \cup E_{ex}(v_i, v_j)$, let $V(e_d) \leftarrow V(e_d) - \{v_i, v_j\}+\{v_n\}$ and then reselect its candidate.

(4) If either v_i or v_j is a boundary vertex, the boundary vertex is replaced by v_n.

[EXAMPLE]

An example shown in Fig. 4.1.4 explains the execution of the vertex combination algorithm. In the hypergraph, there are six vertices indexed by $\{v_1, ..., v_6\}$ and the hyperedge sets $E = \{e_1, ..., e_7\}$, where $v(e_1) = \{v_1, v_2\}$, $v(e_2) = \{v_1, v_2\}$, ..., and $v(e_7) = \{v_5, v_6\}$. There are five vertex combination passes during the algorithm execution. Here, only the first and the last passes are described.

[First Pass]

The candidate set is {{v_1, v_2}, {v_1, v_3}, {v_2, v_4}, {v_3, v_5}, {v_4, v_6}, {v_5,v_6}}, and IOC(v_1, v_2) = 0; IOC(v_5,v_6) = 2; IOC(v_1, v_3) = IOC(v_4, v_6) = 4; IOC(v_3, v_5) = 5; IOC(v_2, v_4) = 6.

(1) SELECTION phase : The vertex-pair {v_1, v_2} is selected for its smallest value of IOC.

(2) PLACEMENT phase : In the case, the ordering is <v_1, v_2>.

(3) REDUCTION phase : Here, v_1 and v_2 are combined to form a cluster v_1'. In the stage of cluster tree construction, v_1 and v_2 are linked to the parent node v_1', where v_1 and v_2 are the left and the right children, respectively as shown in Fig. 4.1.4(b). The shrunk hypergraph then is V = {v_1', v_3, v_4, v_5, v_6}; v_1' = <v_1, v_2>; E = {e_3, e_4, e_5, e_6, e_7}, where V(e_3) = {v_1', v_3}, V(e_4) = {v_1', v_4}, V(e_5) = {v_3, v_4, v_5}, V(e_6) = {v_4, v_6}, and V(e_7) = {v_5, v_6}. The candidate set = {{v_1', v_3}, {v_1', v_4}, {v_3, v_5}, {v_4, v_6}, {v_5, v_6} }.

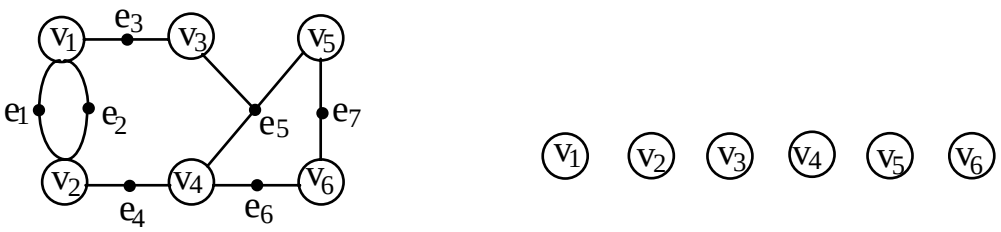
[Final Pass]

V = {v_1", v_5"}; E = {e_4, e_5}, where V(e_4) = {v_1", v_5"} and V(e_5) = {v_1", v_5"}; v_1" = <v_2, v_1, v_3> and v_5" = <v_4, v_5, v_6>.

(1) SELECTION phase : The selected vertex-pair is {v_1", v_5"}.

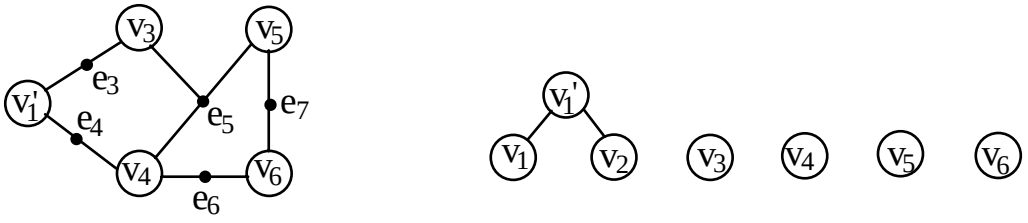
(2) PLACEMENT phase : Both the topological coordinate of e_4 in v_1" and in v_5" are (1, 1). The topological coordinate of e_5 in v_1" and in v_2" are (3, 3) and (1, 2), respectively. Four ordering types, RL, LR, RR and LL, must be considered, where RL(v_1", v_5") = LL(v_1", v_5") = 4 and RR(v_1", v_5") = LR(v_1", v_5") = 7. Since both RL(v_1", v_5") and LL(v_1", v_5") have the smallest connection cost, we can randomly select either one as the ordering type. Here, the ordering type LL(v_1", v_5") is selected.

(3) REDUCTION phase : In cluster tree construction, the algorithm first reverses the vertex ordering in cluster tree v_1". Then, cluster trees v_1" and v_5" are linked to parent node v_1"' as shown in Fig. 4.1.4(f). In hypergraph reduction, v_1" and v_5" are combined to form v_1"'. Since the hypergraph is reduced to only one vertex, the algorithm stops. The vertex ordering obtained is <v_3, v_1, v_2, v_4, v_5, v_6>.



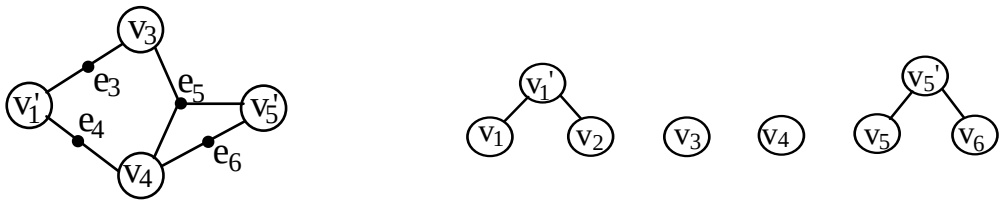
Hypergraph Cluster Tree

(a) An sample hypergraph.



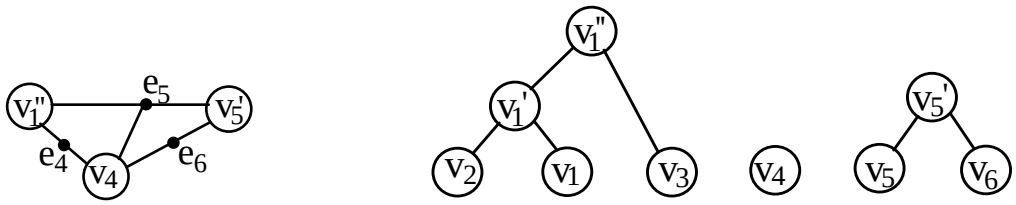
Hypergraph Cluster Tree

(b) The v_1 and v_2 are combined and placed as $v_1' = <v_1, v_2>$.



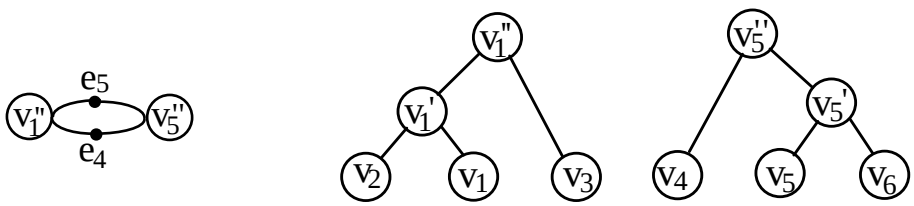
Hypergraph Cluster Tree

(c) The v_5 and v_6 are combined and placed as $v_5' = <v_5, v_6>$.



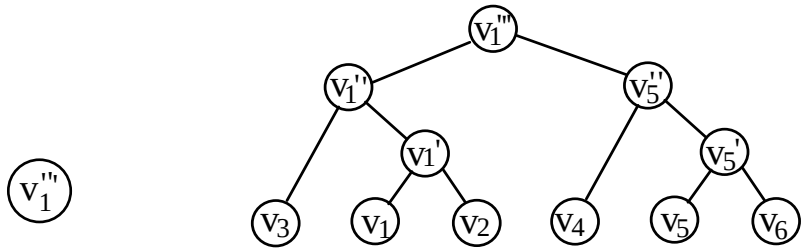
Hypergraph Cluster Tree

(d) The v_1' and v_3 are combined and placed as $v_1'' = <v_2, v_1, v_3>$.



Hypergraph Cluster Tree

(e) The v_4 and v_5' are combined and placed as v_5" = <v_4, v_5, v_6>.



(f) The v_1 "and v_5" are combined and placed as v_1''' = <v_3, v_1, v_2, v_4, v_5, v_6>.

Fig. 4.1.4 The execution of vertex combination algorithm for a sample circuit.

4.1.3 **Complexity Analysis**

4.1.3.1 **Space Complexity**

Basically, the procedure of the vertex combination algorithm is a sequence of cluster tree construction and hypergraph reduction. Thus, the memory space needed is the sum of the size of the initial hypergraph and the final cluster tree. So, its space complexity is O(m).

4.1.3.2 **Time Complexity**

Two measures are used to analyze time complexity. The first is the size of the hypergraph H(V, E). The second, $|e(v_j)|$ is the degree of v_j. By applying Rent's Rule, the measure of the degree of a cluster can be replaced by its size. Rent's Rule has the form of

$$|e(v_i)| = p \times |v_i|^r \tag{4.7}$$

where p is the average degree of the atoms inside the cluster v_i, and r is the Rent exponent. To simplify the analysis, we assume p and r to be constant.

Two binary trees, depicted in Figs. 4.1.5(a) and (b), represent the two extremes of hierarchical structures of the vertex combination processes. The number in each node denotes the size of the corresponding cluster. Each internal node and its two children represent a vertex combination which combines the two children to form the father. In Fig. 4.1.5(a), the vertices of the largest size and of unity size are selected to be placed and combined for each pass; in Fig. 4.1.5(b), two vertices of the same size are selected. Without loss of generality, we can assume that $m = 2^h$, where h is the height of the fully binary tree in Fig. 4.1.5(b).

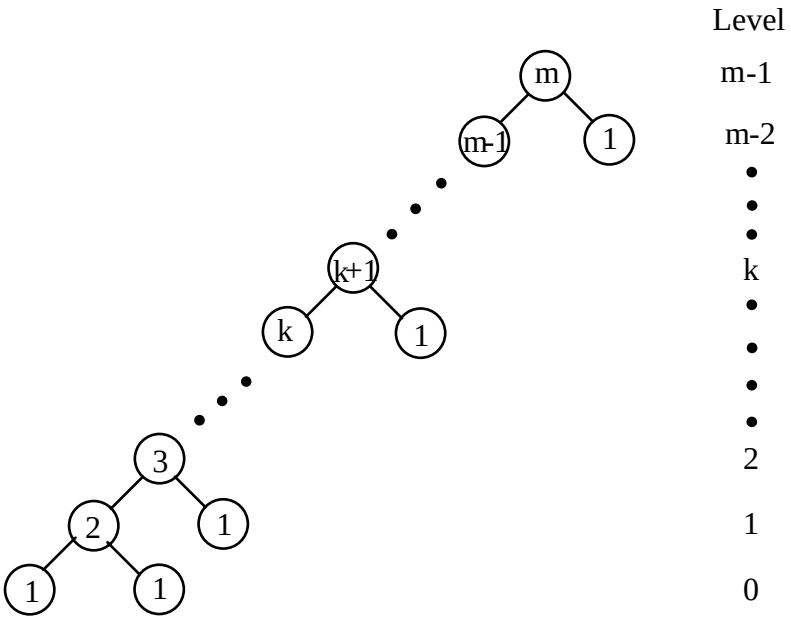
The following lemmas are useful in time complexity analysis.

Lemma 1: Dividing $E(v_i, v_j)$ into three subsets of $E_{in}(v_i, v_j)$, $E_{pi}(v_i, v_j)$, and $E_{ex}(v_i, v_j)$ needs $O(|e(v_i)| \times |e(v_j)|)$ time.

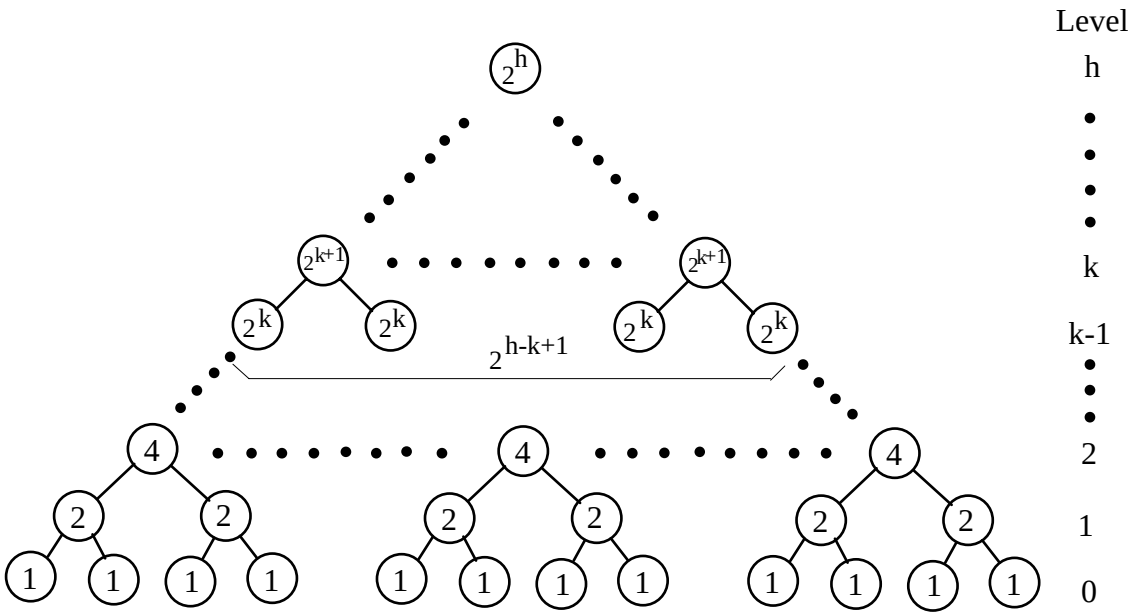
Proof : The operator can be executed by comparing hyperedges in $e(v_i)$ to those in $e(v_j)$, thus it takes $O(|e(v_i)| \times |e(v_j)|)$ time.

Lemma 2 : Calculating $IOC(v_i, v_j)$ takes $O(|e(v_i)| \times |e(v_j)|)$ time.

Proof : Basic operator of calculating $IOC(v_i, v_j)$ is to divide $E(v_i, v_j)$ into three subsets of $E_{in}(v_i, v_j)$, $E_{pi}(v_i, v_j)$, and $E_{ex}(v_i, v_j)$. According to Lemma 1, it needs $O(|e(v_i)| \times |e(v_j)|)$ time.



(a) The combination process with maximum height.



(b) The combination process with minimum height.

Fig. 4.1.5 Two extremes of vertex combination process. The number in a node denote the size of the cluster tree with root of the node.

Lemma 3 : Placing a vertex-pair $\{v_i, v_j\}$ into the right order requires $O(|e(v_i)| \times |e(v_j)|)$ time.

Proof : In the function, we have to calculate and compare the connection cost for all possible types of ordering. Basic operator of calculating the connection cost for each ordering type is to divide $e(v_i, v_j)$ into three sub sets of $E_{in}(v_i, v_j)$, $E_{pi}(v_i, v_j)$, and $E_{ex}(v_i, v_j)$. According to Lemma 1, it takes $O(|e(v_i)| \times |e(v_j)|)$ time. Thus prove the lemma.

(a) **Time Complexity for the Skew Tree**

Consider the vertex combination process at level k. There are $m-k+1$ vertices and about the same size of hyperedges in the current hypergraph. The selected vertex-pair is of size k and 1, respectively. To select the vertex-pair in the candidate set takes $O(m-k+1)$ time. Placing the vertex-pair into right order takes $O(k^r)$ time. In the cluster tree construction, it needs to reverse the ordering of constituent vertices in the cluster trees before joining them together, which takes $O(m-k+1)$ time in the worst case. In hypergraph reduction, there are $O((k+1)^r)$ hyperedges to be updated. For each updated hyperedge, it takes $O(k^r)$ time to reselect its candidate. Thus the REDUCE function takes $O(k^{2r})$ time. The total time complexity then is

$$\sum_{k=1}^{m} \left[O(m-k+1) + O(k^r) + O(k^{2r}) \right] \approx O(m^2 + m^{2r+1}) \tag{4.8}$$

(b) **Time Complexity for the Fully Tree**

Consider a vertex combination process at level k. Both vertices to be selected are of the size 2^k. To select the vertex-pair in the candidate set takes $O(2^{h-k+1})$ time, and to place the vertex-pair takes $O(2^{2rk})$ time. In REDUCTION phase, it takes at most $O(2^{2rk})$ time to rearrange the vertex orderings in the two cluster trees. There are $O(2^{r(k+1)})$ hyperedges to be updated. For each hyperedge, it takes $O(2^{2rk})$ time to select the candidate. Thus, the REDUCE function takes $O(2^{3rk})$ time. The total time complexity then is

$$\sum_{k=1}^{h} 2^{h-k+1} \times \left[O(2^{h-k+1}) + O(2^{2rk}) + O(2^{3rk}) \right] \approx O(M^{3r}) \tag{4.9}$$

(c) **Total Time Complexity**

From the above considerations, we can conclude that the total time complexity of the present algorithm is $O(m^2+m^{2r+1}+m^{3r})$. According to the study by Landman and Russo [Lan71], $0.47 \le r \le 0.75$, so time complexity of vertex combination algorithm is $O(m^{2.5})$. However this is the worst-case estimate. In the following experiments, we will demonstrate the efficiency of this algorithm in terms of CPU running time.

4.1.4 **Experimental Results and Discussion**

The vertex combination algorithm has been implemented in C language, and runs on a SUN SPARC I work-station. To compare it to others, we specifically implemented the cluster growth algorithm proposed in [Kan83] on the same computer.

Three examples from [Che87] are used to test vertex combination algorithm. The results are compared with those of three constructive algorithms: cluster growth method [Kan83], graph partitioning method [Che87], and global method [Che84]. In order to understand the consequences of placement using different objective functions for VLSI design applications, we list both results published in [Kan83, Che87] and vertex combination algorithm in Table 4.1.1 in terms of three different measures: total routing length, tracks required, and CPU running time.

Example one contains 9 vertices and 16 nets. We fix vertices two and six following the result of [Che87]. In the example all methods compared generate the same result.

Example two contains 16 vertices and 17 nets. For comparison purposes, vertices one and six are fixed at the two ends. The ordering obtained by vertex combination algorithm is shown in Fig. 4.1.6 with the layout of tracks. From Table 4.1.1, one can see that vertex combination algorithm yields the layout of the minimum tracks required. The cluster growth algorithm generates an ordering with the same tracks required as vertex combination algorithm, but has a larger total routing length.

Table 4.1.1 : **The comparison between vertex combination algorithm and other algorithms.**

Algorithms	Ex1: 9 vertices, 21 nets, and 57 pins			Ex2: 16 vertices, 17 nets, and 42 pins			Ex3: 31 vertices, 33 nets, and 79 pins		
	Tracks Required	Wire Length	CPU (sec.)	Tracks Required	Wire Length	CPU (sec.)	Tracks Required	Wire Length	CPU (sec.)
Vertex Combination	11	50	< 0.1*	7	77	0.1*	4	90	0.4*
Cluster Growth	11	50	< 0.1*	7	83	< 0.1*	5	96	< 0.1*
Graph Partitioning	11	50	2.1**	8	73	2.7*	5	91	7.3**
Global Method	11	50	1.6**	8	79	1.7*	8	102	4.6**

* Run on a SUN SPARC I work-station.

** Run on a VAX 11/780 machine.

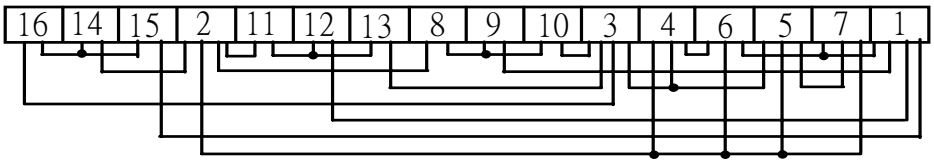


Fig. 4.1.6 Layout of the 16-vertex example

Example three contains 31 vertices and 33 nets. The ordering obtained by vertex combination algorithm is OP1-U-OP3-W-V-X-OP2-Y-Z-P-T-S-O-N-L-K-M-Q-R-E-D-F-I-J-P2-H-G-C-A-B-P1, which is the best placement both in the total routing length and the tracks required when compared to others in the table.

From the above results, it can be concluded that, among those algorithms compared, vertex combination algorithm gives the best placement results.

It is difficult to compare the CPU running time among different algorithms, since they are implemented in different programming languages and run on different machines. However, the cluster growth algorithm seems the fastest according to the executed time for the above three examples. Therefore, we choose it as a reference to compare its running speed with vertex combination algorithm using our SPARC I work-station.

In Table 4.1.2, we list the results by vertex combination and cluster growth algorithms for nine randomly chosen circuits. It can be seen that vertex combination algorithm gives better results both in tracks required and in total routing length. Both of them are about one-third those obtained by cluster growth algorithm for vertex numbers greater than a value of approximately 500. The relationship between the vertex number of the original hypergraph, m, and the total routing length L is shown in Fig. 4.1.7 together with the best curve-fitted equations. One can observe that L is proportional to $1.26\times m^{1.3}$ and $2.1\times m^{1.4}$

Table 4.1.2: **Comparisons between vertex combination and cluster growth algorithms**.

Circuits	Vertex Number	Net Number	Vertex Combination Algorithm			Cluster Growth Algorithm		
			Tracks Required	Wire Length	CPU Time(sec)	Tracks Required	Wire Length	CPU Time(sec)
A	44	26	6	184	0.1	8	221	< 0.1
B	61	38	6	264	0.5	8	287	0.1
C	294	186	13	2266	1.1	29	3862	3.3
D	524	308	19	5278	3.1	58	14519	28.7
E	533	313	13	3759	2.7	61	16427	23.2
F	614	374	14	5443	2.8	46	15906	28.0
G	674	415	14	6103	3.3	64	20509	36.6
H	786	489	20	8555	4.0	55	26251	48.9

I	1453	922	21	17615	9.6	60	55840	133.5

for the vertex combination and cluster growth algorithms respectively. Thus, as m increases, vertex combination gives a much smaller routing length. Furthermore, as shown in Fig. 4.1.8, the CPU time, T, is proportional to m for vertex combination algorithm, whereas it is proportional to $m^{1.8}$ for cluster growth algorithm. Therefore, except for small-sized problems, the vertex combination algorithm is faster than cluster growth algorithm.

Why vertex combination algorithm can be more efficient and faster than the cluster growth algorithm ? It is because vertex combination algorithm considers the net interconnection at a global level in the SELECTION phase, thus it can avoid the traps of local optima. In contrast, the cluster growth algorithm only selects a vertex-pair at a local level, thus can easily falls into local optima. Comparing the execution time, vertex combination algorithm takes more time than the cluster growth algorithm in the phases of SELECTION and PLACEMENT. But it takes much smaller time in the REDUCTION phase, since the average number of nets incident upon a selected vertex-pair is fewer in vertex combination algorithm than that in the cluster growth algorithm, especially for large-sized problems. Thus, vertex combination algorithm is faster in large-sized problems.

In order to show the uniformity of the vertex combination algorithm, the congestion analysis for Circuit I in Table 4.1.2 is depicted in Fig. 4.1.9 as a typical example. The distribution of cut values has two trends. One is a relatively crowded region with larger cut values. The other is a relatively non-crowded region with smaller cut values. Obviously, most of the cut values fall into the crowded region for cluster growth algorithm, but the cut values for vertex combination algorithm are in the non-crowded region. The trend indicates that verttex combination algorithm gives a placement with a net distribution more uniform than that provided by the cluster growth algorithm.

4.1.5 Summary

The linear placement problem plays an important role in the growing application of modern PCB layout and VLSI designs. Since the goal of an initial placement is to obtain a moderate performance placement in a smaller amount of CPU time, it is desirable that the algorithm is efficient and fast. In this study, we present a constructive algorithm, vertex combination algorithm, to meet those requirements.

The objective function of the vertex combination algorithm is the minimization of both the total routing length and the maximum-cut-value. It reflects the routability of signal nets more precisely than those use only the objective of either minimum total routing length or the min-cut.

Vertex combination algorithm considers global net interconnections in the SELECTION phase as top-down heuristics, thus avoids falling into the traps of local optima. In the REDUCTION phase, it reduces the size of the problem one at a time as bottom-up approaches, thus it is very fast. Experimental results confirm that it performs better than

those of several previously known popular algorithms, both in the total routing length and the execution time.