Acknowledgment

I would like to thank my guide, Prof. Sachin Patkar, for all the guidance and support. This report is result of his consistent encouragement and resourceful inputs. I am also thankful to all my friends, colleagues and family members for their support and encouragement.

Contents

1 Introduction **5**
1.1 Physical Design: 5
1.2 Algorithmic techniques: 6

2 Network Flow Problems **8**
2.1 Maximum flow Definition 8
2.2 Ford-Fulkerson Method 8
2.3 Push-relabel algorithm 10

3 Floor Planning **13**
3.1 Introduction 13
3.2 Hierarchical Approach 14
3.2.1 Bottom-up approach: 14
3.3 Floorplan Sizing 15
3.3.1 Floorplan Sizing for Fixed cells: 15
3.3.2 Hierarchical Floorplan Sizing for Variable cells: 16

4 Placement **18**
4.1 Placement by Simulated Annealing 18
4.1.1 Standard Cell Placement by SA: 19
4.2 Placement by Genetic Algorithm 20
4.2.1 Macro Cell Placement by GA: 22
4.3 Regular Placement: Assignment Problem 23
4.4 Module Placement Based on Resistive Network 24
4.4.1 Network Analogy 25
4.4.2 Proposed Method 26
4.5 Timing Driven Placement 27
4.5.1 Zero Slack Algorithm 27

5 Routing **31**
5.1 Fundamentals 31
5.1.1 Maze Running 31
5.1.2 Steiner Trees 31
5.2 Global Routing 33
5.2.1 Randomized Routing 34
5.3 Detailed Routing 35
5.3.1 Channel Routing 35
5.4 Detailed Routing Of Standard Cells Using Side Channel 37

6 Timing Improvisation **39**
6.1 Introduction . . . 39
6.2 Calculation of buffer insertion segments . . . 39
6.2.1 Algorithm to calculate the cost of the segments in the critical path . . . 39
6.2.2 Algorithm to calculate the segments with lowest cost but within specific distance in the critical path . . . 41

7 Partitioning **43**
7.1 Partitioning Problem Formulation: . . . 43
7.2 A New Partitioning Approach: . . . 43
7.2.1 Path Enumerating Algorithm . . . 44
7.2.2 Partitioning by Simulated Annealing . . . 45
7.2.3 Partitioning by r-balanced Flow Method . . . 46
7.2.4 Partitioning by Branch and Bound method . . . 50

8 Conclusion **52**

9 Future Work **53**

10 Results **54**
10.1 Optimal area implementation for a sliceable floor plan . . . 54
10.2 Placement of Standard cell by Simulated Annealing . . . 55
10.3 Placement of Macro cell by Genetic Algorithm . . . 56
10.4 Placement by Regular Placement Assignment Problem . . . 56
10.5 Zero Slack Algorithm . . . 58
10.6 Steiner tree . . . 60
10.7 Randomized Routing . . . 61
10.8 Left Edge Algorithm . . . 62
10.9 Detailed Routing of Standard Cell with Side Channel . . . 63
10.10 Calculation of Buffer Insertion Segments in the critical path . . . 64
10.11 Example of Partitioning a Circuit with New Approach . . . 65
10.12 Clustering by Simulated Annealing, r-balanced Flow Based Method and Branch and Bound Method . . . 68

List of Figures

2.1 Example of Maximum flow calculation by Ford Fulkerson method 9
2.2 Maximum flow calculation by push-relabel algorithm 12

3.1 Hierarchical floor plan of order 5 13
3.2 A sliceable floorplan and its corresponding tree. 14
3.3 A smallest nonsliceable floorplan. 14
3.4 (a) Circuit connectivity graph (b)Tree constructed by greedy method (c) Corresponding Floorplan 15
3.5 (a) Fixed Floorplan (b) Maximal vertical segments (c) Horizontal dependency graph 16
3.6 A horizontal clustering of a floorplan 17

4.1 (a) Arrangement of positions (b) Module and associated position number for initial solution (c) New solution by interchanging first two modules. 20
4.2 Flow chart of simple Genetic Algorithm 21
4.3 (a) Arrangement of positions. (b)and(c) Two solutions in the initial population. 22
4.4 Crossover at a random point to generate two offsprings from two parents 23
4.5 An n terminal passive resistive network where first m nodes are floating 25
4.6 (a) Computation of arrival time at an input pin (b) Computation of required time at an output pin 28
4.7 Zero slack algorithm example, Step:1 29
4.8 Zero slack algorithm example, Step:2 29
4.9 Zero slack algorithm example, Last Step 30

5.1 Labeling by the Lee-Moore algorithm with one shortest path 32
5.2 Construction of Steiner tree 33
5.3 (a) A channel with terminal list (b) Corresponding vertical constraint graph . . . 36
5.4 Example of result of left edge algorithm 37
5.5 Detailed routing of standard cells using side channel 38

6.1 Input graph 40
6.2 Constructed flow graph G_f from G, for the segment $1-2$. 40
6.3 A subgraph G_{sub}, which includes only the critical path, with calculated cost on the edges 41
6.4 Graph G_{br}, which consists of red and blue edges 41
6.5 Modified Graph of G_{br} 42

7.1 (a) Original Graph. (b) Graph after the modification of flip-flop nodes. 45
7.2 Graph G and its corresponding flow graph G_f. 48
7.3 A typical flow graph when max-flow is reached 49
7.4 A solution space structure for partitioning a set of four vertices 50

Chapter 1

Introduction

Automation of various steps involved in the design and fabrication of integrated circuits has helped to enhance the growth of integration technology. Automation of a given design process requires a mandatory algorithmic analysis. The project aims at exploring the various algorithms in VLSI physical design, which serve as a basis for the research and development of new Computer Aided Design (CAD) tools.

1.1 Physical Design:

In the physical design process, the netlist created from the Register Transfer Level (RTL) description is converted to a geometric description, called layout. These techniques aim to produce layouts with a small area, since cost of fabricating a circuit is a function of the circuit area. Also other optimality constraints like wire length minimization, delay minimization etc to be considered.

Since physical design is a complex optimization problem involving several objective functions and large number of components, CAD tools have been developed to facilitate the design process. It is accomplished in several stages such as partitioning, floor planning, placement and routing.

- Partitioning: It is the task of dividing the circuit into smaller parts, which can be used for placement, floor planning etc. It allows you to work with smaller slices of data .The main objective of partitioning is to balance the size of blocks and reduce the interaction between the blocks.

- Floor planning: Floorplanning is the determination of the approximate location of each module in a rectangular chip area. Various floorplanning approaches has been proposed based upon the connectivity between the modules. Routing space can be reduced by placing highly connected modules close to each other.

- Placement: The process of placement involves ascertaining the best position for each module on the chip according to the appropriate cost function. In this process, the shape and terminal location of each module is fixed.

- Routing: The objective of the routing phase is to complete the interconnections between the placed modules according to the specified netlist. Routing is usually done in two phases, referred to as Global Routing and Detailed Routing. Global routing only specifies the different regions in the routing space through which a wire should be routed. Detailed routing uses the information obtained by the global router to decide the exact location of the interconnects within the wiring area.

1.2 Algorithmic techniques:

Most of these layout problems can be solved exactly, only with exponential time. But since it is not affordable, many fast and good quality algorithmic techniques have been designed. Some of these techniques are the following.

- Greedy approach: Greedy algorithms are simpler than other classes of algorithms but do not always produce globally optimal solutions. These algorithms go through a series of step and each step a choice is made, that gives a locally optimal solution.
- Dynamic programming: In this approach, a problem is solved, first by partitioning into a collection of subproblems and then by combining these solutions. It is applied when the subproblems are not independent. It solves every subproblem just once and saves its answer in a table. This avoids the work of recomputing the answer every time the subproblem is encountered. Generally it work backwards from the last decision to the earliest one.
- Hierarchical approach: In this approach also, a problem is partitioned into a set of subproblems but subproblems are independent. Usually solution is constructed in a bottom up fashion.
- Linear programming: In this approach the objective function is a minimization or maximization problem subject to a set of constrains expressed as a collection of inequalities.
- Simulated annealing: Simulated annealing algorithm is a technique used to solve general optimization problems based on the annealing process in crystals. It evaluates the set of feasible solutions of the problem in sequence, moving from one solution to another with some probability.
- Genetic algorithms: Genetic algorithms are computational procedures that mimic the natural process of evolution. It works by evolving a population of solutions over a number of generations. For each generation, solutions are selected from the population based on the fitness value. These solutions by crossover (merging previous solutions) and by mutation (modifying the solutions) generates new population.
- Branch and Bound: Branch and Bound algorithm systematically searches the complete space of solutions for a given problem for the best solution. A tree structured configuration space can be used to avoid searching the entire solution space, by stopping at a node which does not lead to an optimal solution.

This report will look at the above algorithmic techniques in the important stages of VLSI physical design. Hierarchical bottoming up approach for floorplanning, simulated annealing, genetic algorithm and linear programming approach for placement are explained in detail. We have used, *Scilab* for implementing simulated annealing, and *LP Solver* for linear programming. A method based on resistive network analogy of the placement process is also explained.

Randomized routing, a global routing algorithm, which uses integer programming technique, is implemented with the help of *LP Solver*. We have developed an algorithm for the detailed routing of standard cell with side channel. Here left edge algorithm is used for the routing of each channel.

A timing improvisation method, where the delay of the critical path can be reduced, by inserting buffers in some specific critical path segments is also explained. The segments which are not inside dense clusters, but within specific distance apart, are considered.

We have also developed a new approach for circuit partitioning. In this approach a graph, where the nodes represent flip-flops and the edges represent their connectivity, is constructed. Since, the size of the graph is small as compared to the original graph, many efficient partitioning approaches can be applied with less complexity, which will finally give a partition for the original circuit.

Many processes in VLSI physical design can also be modeled as a network flow problem. Maximum flow or minimum cut can be calculated, by performing Ford-Fulkerson or Push-Relabel algorithm on the flow network, which gives the required solution.

With the development of Field Programmable Gate Arrays (FPGA) new CAD algorithms are required to make effective use of logic and routing resources. The algorithmic techniques and the data structures discussed in this report will facilitate the research and development of physical design algorithms on FPGAs.

Chapter 2

Network Flow Problems

2.1 Maximum flow Definition

We can interpret a directed graph as a flow network, to model many problems in VLSI physical design. A *flow network* $G = (V, E)$ is a directed graph in which each edge $(u, v) \in E$ has a nonnegative capacity $c(u, v) \geq 0$. If the edge $(u, v) \notin E$, then we define $c(u, v) = 0$. There are two special vertices called *source* and *sink* and each vertex $v \in V$ is on some path from source s to sink t, thus the graph is connected.

A flow f, in G is a real valued function, that satisfies the following two properties:

$$For\ all \quad u, v \in V, \quad 0 \leq f(u, v) \leq c(u, v).$$

where, $f(u, v)$ is the flow from u to v. This is called as *capacity constraint*.

The second property, called as *flow conservation* is defined as,

$$For\ all \quad u \in V - \{s, t\}, \quad \sum_{v \in V} f(v, u) = \sum_{w \in V} f(u, w)$$

where, v and w represents the fan-in and fan-out nodes of u respectively.

The *value* of a flow is defined as the total flow out of the source minus the flow into the source. The *maximum flow problem* is to find a flow of maximum value, for a flow network G, with source s and sink t.

2.2 Ford-Fulkerson Method

This method depends upon three main ideas called residual network, augmenting paths, and cuts. In this method, initial flow value of *zero*, iteratively increasing, by finding an augmenting path in the associated residual network. Both the *capacity constraint* and *flow conservation* properties are maintained in each iteration.

Residual networks: The residual network, G_f, is constructed from the flow network G and the given flow f. From G, calculate the possible increase and decrease of a positive flow for each edge, which is represented by forward and reverse edges of G_f, respectively. The possible increase of a positive flow is $c(u, v) - f(u, v)$ while possible decrease of a flow is $f(u, v)$. Flow in the residual network is defined with respect to *residual capacities*, c_f.

For a flow network of $G(V, E)$ with $u, v \in V$ the *residual capacity* is defined by,

$$c_f(u, v) = \begin{Bmatrix} c(u, v) - f(u, v) \;\; if(u, v) \in E, \\ f(v, u) \;\; if(v, u) \in E, \\ 0 \;\; otherwise \end{Bmatrix}$$

Augmenting path: An augmenting path is a simple path from s to t in the residual network G_f, where the flow can be increased. We can increase the flow on an edge (u, v) of an augmenting path by up to $c_f(u, v)$, without violating the capacity constraint. The maximum amount by which the flow can be increased on each edge in an augmenting path p is the *residual capacity* of p and is given by,

$$c_f(p) = min \{ c_f(u, v) : (u, v) \; is \; on \; p\}.$$

Cut of flow network: A $cut(S, T)$ of flow network $G = (V, E)$ is a partition of V into S and $T = V - S$ such that $s \in S$ and $t \in T$. The *capacity* of the cut is defined by,

$$c(S, T) = \sum_{u \in S} \sum_{v \in T} c(u, v).$$

If the capacity of the cut is minimum over all cuts of the network, it is called as *minimum cut*. For a flow f, the *net flow* $f(S, T)$ across the cut (S, T) is defined to be,

$$f(S, T) = \sum_{u \in S} \sum_{v \in T} f(u, v) - \sum_{u \in S} \sum_{v \in T} f(v, u).$$

The net flow across any cut is same as the value of the flow, $| \; f \; |$, and is bounded from above by the capacity of any cut of G. Thus $| \; f \; | \leq c(S, T)$ for all cuts (S, T).

If the residual network has no augmenting path, flow f is maximum. Thus, $| \; f \; | = c(S, T)$ for some cut (S, T) of G, and that cut will be a *minimum cut*. The theorem which says that the value of a maximum flow is equal to the capacity of a minimum cut is called as $max - flow \; min - cut$ theorem.

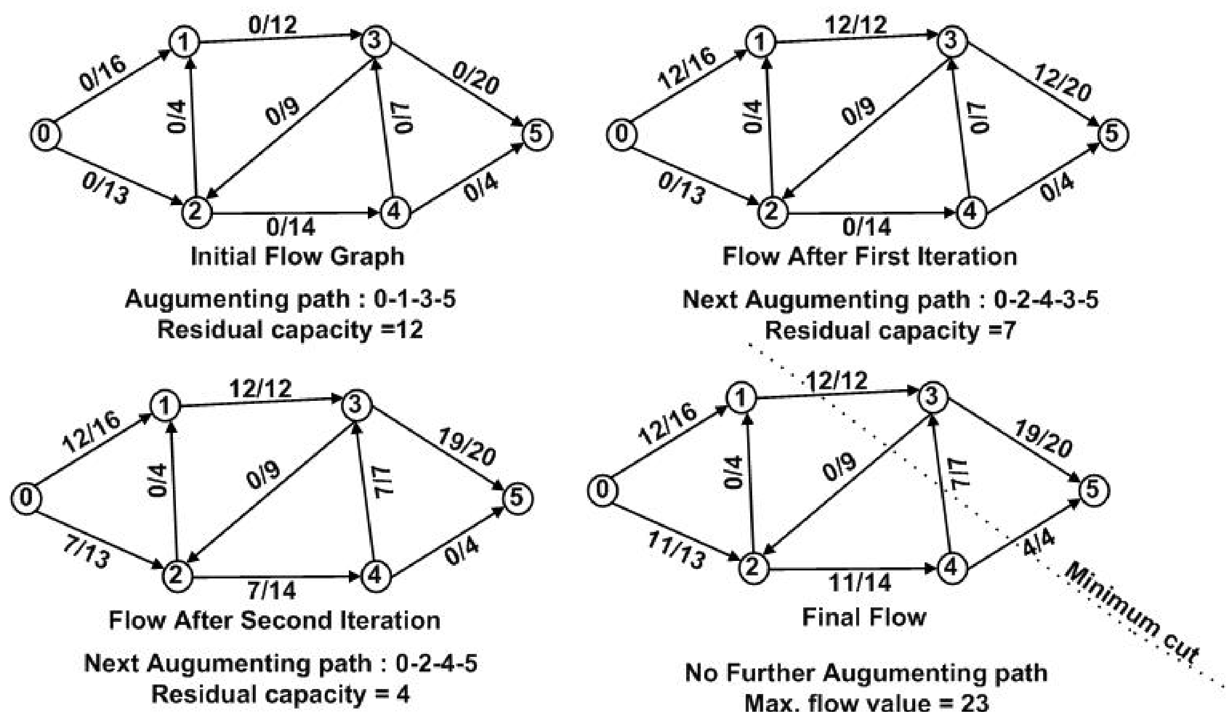

Figure 2.1: Example of Maximum flow calculation by Ford Fulkerson method

The basic Ford-Fulkerson algorithm: In each iteration, it calculates the augmenting path p and modify the flow f by using p. The algorithm can be summarized as follows:

Steps:

1. Initialize the flow of edge $(u, v) \in G(V, E)$ as 0.
2. Find an augmenting path p in G_f from s to t.
3. Calculate the *residual capacity* of path p by using the formula, $c_f(p) = min \ \{ \ c_f(u, v) : (u, v) \ is \ on \ p\}$.
4. Update the flow in the original network. If the residual edge is an original edge of G, then the $c_f(p)$ is to be added with the current flow, else it to be subtracted.
5. Check for an augmenting path p, if exists, repeat steps $2 - 5$.
6. If there is no augmenting path, then the flow is maximum.

An example is shown in figure 2.1 with nodes 0 and 5 are the *source* and *target* nodes respectively. In the final flow graph, the net flow across any cut is 23. The cut, where the capacity is same as 23 is shown as dotted line, and it is a minimum cut. For this cut, the nodes on the *source* side and *target* side are $\{0, 1, 2, 4\}$ and $\{3, 5\}$ respectively. All the forward edges from source side to target side are saturated and the reverse edges from target side to source side carry zero flow. Thus no augmenting path exists and the flow is maximum.

For a flow network $G(V, E)$, The total number of flow augmentations performed by the algorithm is $O(VE)$. Calculation of augmenting path can be done in $O(E)$ time, by breadth first search. Thus the total running time becomes $O(VE^2)$.

2.3 Push-relabel algorithm

This algorithm works on one vertex at a time, looking only at the vertex's neighbours in the residual network. It satisfies the capacity constraint but not the flow conservation property. A relaxation of flow conservation that the flow into a vertex may exceed the flow out, is maintained. The amount by which the flow in exceeds the flow out is called as *excess flow*. Basically two operations are performed in this algorithm, called *pushing* and *relabeling*. Flow can be pushed only from a higher vertex to a lower vertex. Thus the relabeling operation may be required.

Initially, The height of the source is assigned as $\mid V \mid$ and that of all other vertices as 0. The flow is pushing from the source to its neighbouring vertices, according to the maximum capacity of the corresponding edges. Since these vertices will have excess flow, these to be further pushed, but can be done only after relabeling. By relabeling, height of a vertex is increasing with respect to its neighbours, to which it has an unsaturated edge. If further pushing is not possible in forward direction, then algorithm sends the excess flow in the vertices back to the source. This can be done by relabeling the vertices above the height of the source. If no vertices have excess flow, then the flow is maximum.

Let $G(V, E)$ be a flow network with source s and target t, and f be a preflow in G. Let the height and the excess flow stored at a vertex u is $u.h$ and $u.e$ respectively. The steps to be performed in this algorithm are:

Steps:

1. Initialize preflow as follows.

(a) $f(u,v)$, for all $(u,v) \in E$ as 0.

(b) $u.e = 0$ and $u.h = 0$ for all $u \in V$.

(c) $s.h = \mid V \mid$.

(d) For all the adjoint vertex, v of s, calculate the following.
$f(s,v) = c(s,v)$.
$v.e = c(s,v)$.
$s.e = s.e - c(s,v)$.

2. Check for an active vertex u, where $u.e > 0$. Set the forward push flag, if any edge starting from u is unsaturated.

3. Perform relabel operation as follows:
If the forward push flag is set then,
$u.h = 1 + min\{v.h\}, \ \forall v \ such \ that, f(u,v) < c(u,v)$.
else,
$u.h = 1 + min\{w.h\}, \ \forall w \ such \ that, \ f(w,u) > 0$.

4. Push $u.e$ to its neighbouring vertex, v which satisfies the condition, $u.h = v.h + 1$. Calculate the following,

   (a) The amount of excess flow to be pushed, $\Delta_f(u,v) = min\{u.e \ , \ (c(u,v) - f(u,v))\}$.

   (b) $u.e = u.e - \Delta_f(u,v)$

   (c) $v.e = v.e + \Delta_f(u,v)$

   (d) If the forward push flag is set, then, $f(u,v) = f(u,v) + \Delta_f(u,v)$.
   else, $f(v,u) = f(v,u) - \Delta_f(u,v)$.

5. Check for an active vertex, and repeat step 2-4.

6. If there is no active vertex, then the flow is the maximum flow.

An example of maximum flow calculation by this algorithm is shown in figure 2.2. Initially, the flow value 16 and 13 are pushed from $source$ to $node-1$ and $node-2$ respectively. Now, $1.e = 16$ and $2.e = 13$. Let $node-2$ be the active vertex. Relabel $node-2$ as, $2.h = 4.h + 1$, thus $2.h = 1$. The amount of excess flow can be pushed from $node-2$ to $node-4$ is 13, since $c(2,4) > 2.e$. The updated flow values are, $f(2,4) = 13$, $2.e = 0$ and $4.e = 13$. New the active vertices are $node-1$ and $node-4$. Consider any one and continue the process, till the excess flow of all the nodes are zero. Then the flow is maximum.

Since always the height of source is $\mid V \mid$ and target is 0, at any time during the execution, $u.h \leq 2 \mid V \mid -1$, for all $u \in V$. Thus bound on the total number of relabel operations is $2 \mid V \mid^2$, and that of saturated pushing operation is $2 \mid V \mid\mid E \mid$. The total number of non saturating push is less than $4 \mid V \mid^2 (\mid V \mid + \mid E \mid)$. Thus overall, the algorithm runs in $O(V^2E)$ time [7].

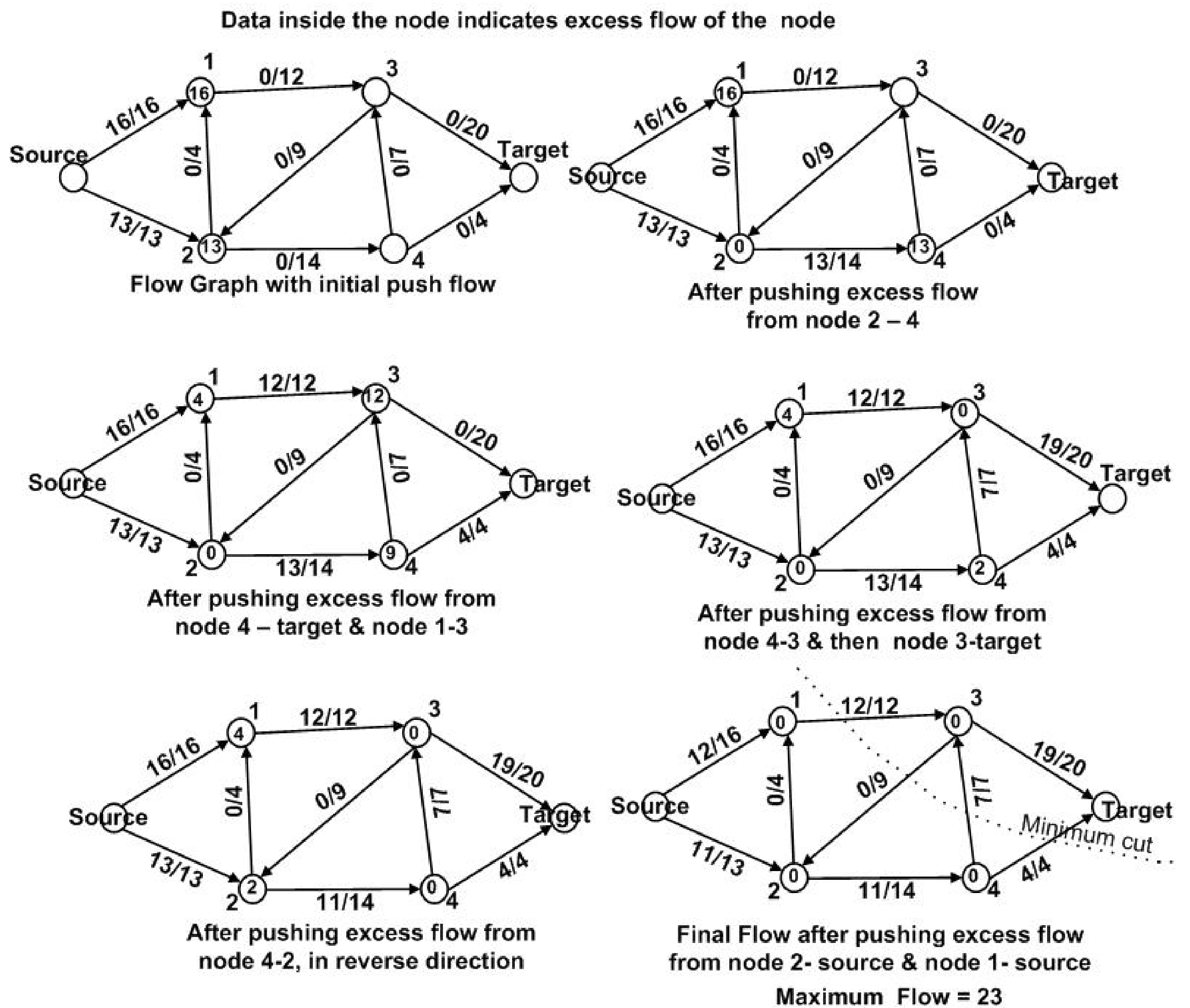

Figure 2.2: Maximum flow calculation by push-relabel algorithm

Chapter 3

Floor Planning

3.1 Introduction

The floor planning problem is to determine the approximate location of each module in a chip area. Main objective of a floor planning algorithm is to minimize the total chip area and the routing space.

Generally, modules with relatively high connectivity placed close to one another, which reduces the routing space. The weight of the nets between the modules can define the closeness of modules. Based upon this measure of closeness various floor planning approaches have been proposed.

A floorplan is usually represented by a rectangle since this is the most convenient shape for chip design. This rectangle is dissected with several vertical and horizontal lines to mark the borders of the modules. Module representation also restricted to rectangles, to facilitate automation. Some of the restricted floorplans and their characteristics will be discussed.

Hierarchical floorplan: It is a floorplan that can be described by a floorplan tree. The leaf nodes of the tree corresponds to modules and internal node defines how its child floorplans are combined to form a partial floorplan. An example of a hierarchical floorplan of order 5 and its floorplan tree is shown in Figure 3.1. Internal node H indicates, leaf nodes 7 & 8 are combined horizontally. Similarly V indicates vertical combination. O_5 represents an implementation of a non-sliceable configuration. This type of floorplan can be used for implementing sliceable as well as nonsliceable floorplans, but

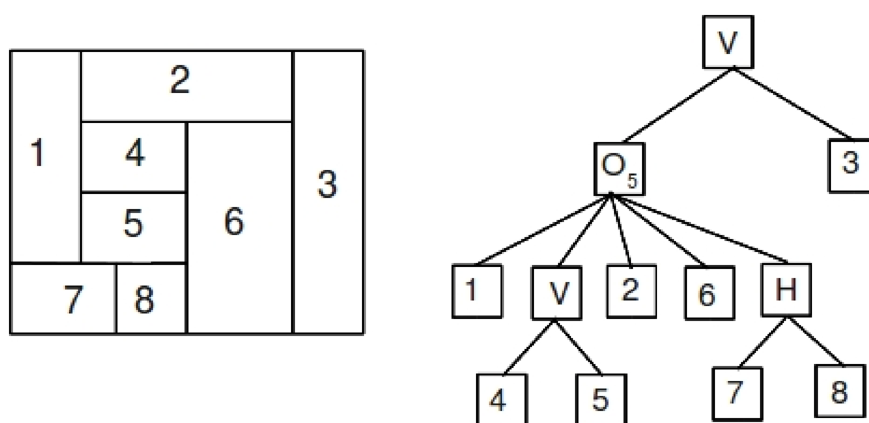

Figure 3.1: Hierarchical floor plan of order 5

with hierarchical frame work. These floorplans are more flexible but computationally more complex.

Sliceable and nonsliceable floorplan: Sliceable floorplan is a special case of hierarchically defined floorplan, and is one of the simplest. It can be represented by a binary tree. It is defined as a floorplan that can be bi-partitioned into two sliceable floorplans with a horizontal and vertical line. Since it has a binary tree structure, it facilitates efficient algorithmic design. A sliceable floorplan and its corresponding binary tree is shown in figure 3.2. A smallest nonsliceable floorplan is shown in figure 3.3.

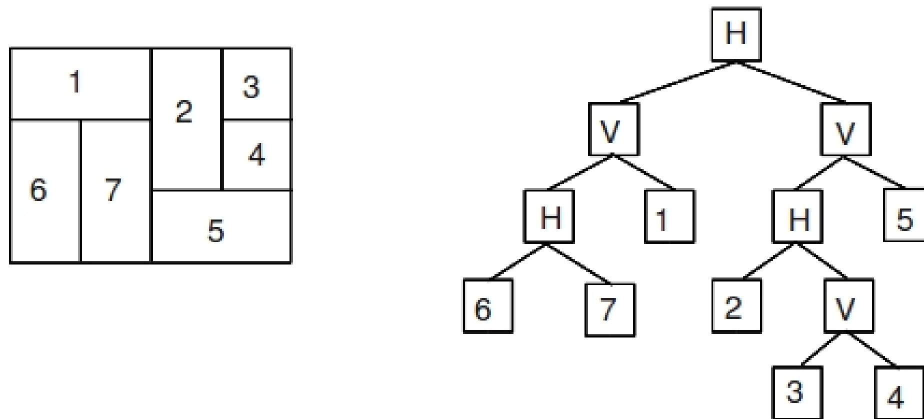

Figure 3.2: A sliceable floorplan and its corresponding tree.

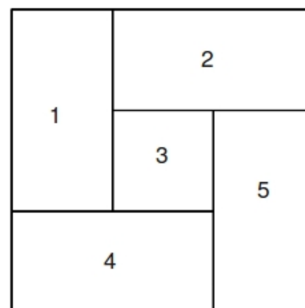

Figure 3.3: A smallest nonsliceable floorplan.

3.2 Hierarchical Approach

In this approach, only a small number of rectangles (modules) are considered at each level of the hierarchy. After an optimal configuration of these rectangles, they are merged into a larger single module. The number of possible floorplans increases exponentially with the number of modules d considered at each level. This approach works best in bottom-up fashion[1].

3.2.1 Bottom-up approach:

Bottom-up approach is based on clustering technique. The modules are represented as vertices in a graph, where the edges represent the connectivity of the modules. Modules with high connectivity are clustered together. Each cluster will have d number of modules. After an optimal floorplan determination, the cluster is merged into a larger module for high level processing.

Clustering can be done by using a greedy procedure in which edges are sorted by decreasing

weights. Choose the heaviest edge, and the two modules of that edges are clustered in a greedy way. Vertices in this cluster are merged and the edge weights are summed up for the next level.

An example is illustrated in figure 3.4 with $d = 2$. Here, since the weight of the edges between a and c are the heaviest, they are combined in vertical fashion, to a single node ac. Similarly b and d also combined in vertical fashion to bd. The connectivity between ac and bd are high as compared to that of e, thus, they are combined in horizontal fashion. Finally, node e is combined with node $abcd$ in vertical fashion. But here, since module e has light weight connections with other modules,

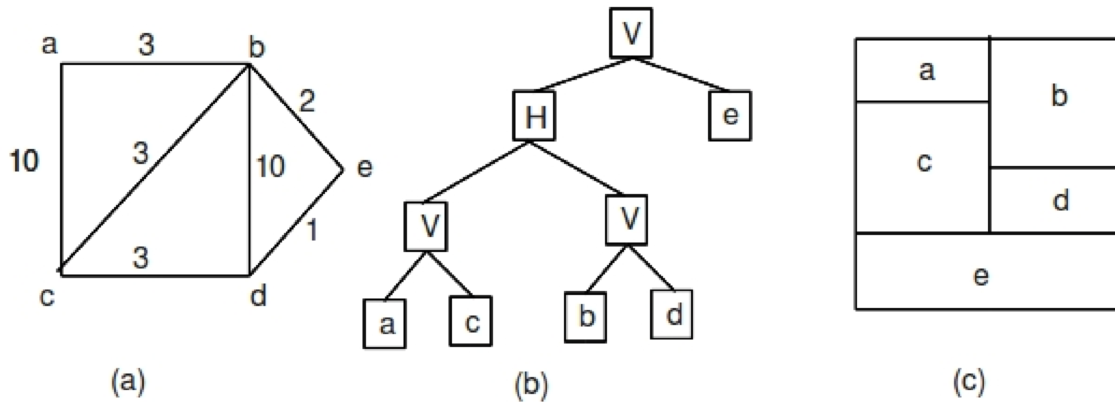

Figure 3.4: (a) Circuit connectivity graph (b)Tree constructed by greedy method (c) Corresponding Floorplan

its edges will be selected at the higher level. If the size of the module e is small, a large area will be wasted in the floorplan.

Floorplans are evaluated at each level, to get the best one, by the cost function. Usually the cost is based on the wiring cost and the area of the floorplan. The wiring cost can be calculated by summing up the multiplication of edge weights and the distance between the centers of the clusters. Area of the floorplan can be estimated from the dimensions, which can be passed up from the bottom up clustering.

3.3 Floorplan Sizing

The floorplan sizing problem is to determine the appropriate module implementation. Each module can have many possible implementations according to different size, speed and power consumption. Poor choice of this may waste a large amount of space. If a cell (module) has only one implementation, it is called as *fixed cell*, otherwise called as *variable cell*.

3.3.1 Floorplan Sizing for Fixed cells:

Area of a fixed cell floorplan, can be calculated by *horizontal and vertical dependency graphs*. A *horizontal dependency graph* A_h(V,E) is a directed acyclic graph, where V is the set of maximal vertical line segments in the floorplan. Each module of the floorplan is associated with a directed edge (v_i,v_j), where v_i and v_j are the vertical segments to the left and right of the module respectively. Length of the edge is the width of the corresponding module in the floorplan. Distance of each vertical line segment (vertex in graph) from the leftmost vertical segment can be calculated. The longest path from the left most vertical segment to the right most vertical segment gives the minimum width of the chip.

An example is shown in Figure 3.5. In this example, module a is represented by an edge from

vertex 1 to 2. Length of this edge is the width of the module a. All other modules are represented in similar way. Minimum width of the chip is the longest path from vertex 1 to 4.

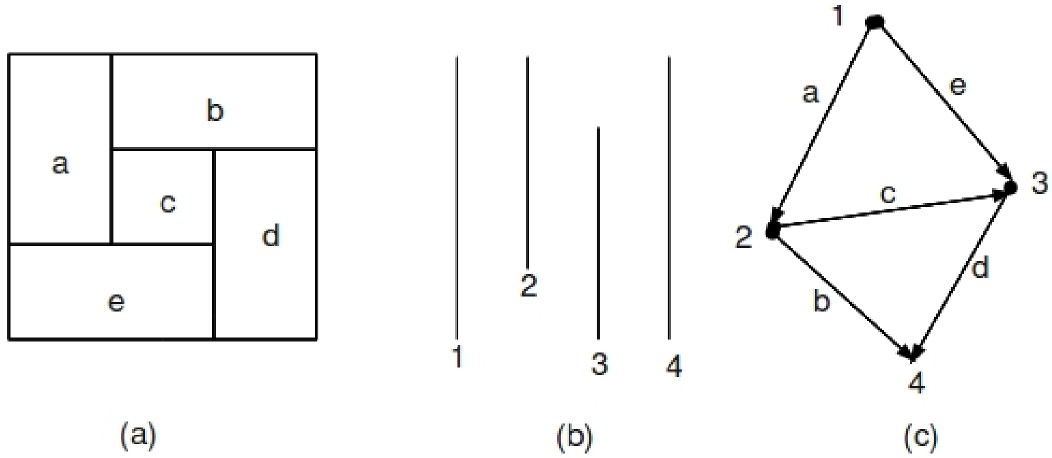

Figure 3.5: (a) Fixed Floorplan (b) Maximal vertical segments (c) Horizontal dependency graph

Similarly, *vertical dependency graph* is used to find the height of the chip. Here consider V, as the set of maximal horizontal line segments in the floorplan. Length of the edge is the height of the corresponding module in the floorplan. The longest path from the bottom most horizontal segment to the top most horizontal segment gives the minimum height of the chip. Since width and height of the chip are calculated, the area of a fixed cell floorplan can be computed easily.

3.3.2 Hierarchical Floorplan Sizing for Variable cells:

Algorithm by Otten and Stockmeyer finds an area optimal implementation of variable cells on a sliceable floorplan [1]. Given, a sliceable floorplan F and a set of possible implementation for each module M_i. Let k^{th} implementation of a module is described by (w_k, h_k), where w_k represents width, and h_k represents height of the module. Let us assume that, the set is sorted, such that,

$$w_1 < w_2 < w_3 ...... < w_{s_i}$$

and

$$h_1 > h_2 > h_3 ...... > h_{s_i}$$

for each module M_i with s_i possible implementations.
Lemma: Given two subfloorplans corresponding to two subtrees of a node u, one with t and the other with s non redundant implementations, then u has at most $s + t - 1$ non redundant implementations.

Proof: Consider an example shown in Figure 3.6. Suppose the possible implementations of left and right subtrees after sorting are $\{(a_1, b_1), (a_2, b_2), ... (a_s, b_s)\}$ and $\{(x_1, y_1), (x_2, y_2), ... (x_t, y_t)\}$, respectively. Then total $s \times t$ implementations are possible, but it is not a practical solution.

Let the left and right subtrees of a node u are implemented by (a_i, b_i) and (x_j, y_j) pairs, then the dimensions of the node is given by $(a_i + x_j, \max(b_i, y_j))$. If $\max(b_i, y_j) = b_i$, then for $k > j$, (a_i, b_i) and (x_k, y_k) implementations can be ignored. Similarly, if y_j is the dominant one, then for $l > i$, (a_l, b_l) and (x_j, y_j) implementations can be ignored. Thus, no need to consider all possible pairing, at each node.

Based on this lemma, if the modules are clustered horizontally at a node, all non redundant implementation of that node can be generated by *vertical node sizing algorithm*.

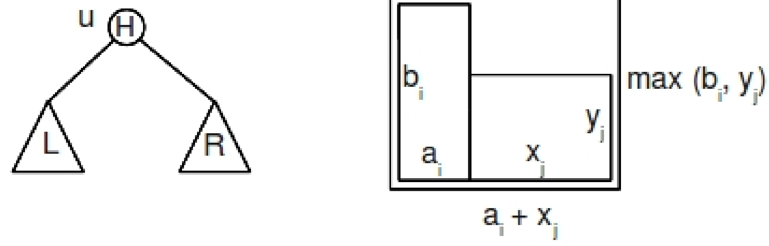

Figure 3.6: A horizontal clustering of a floorplan

Vertical Node Sizing Algorithm:

Input: Two sorted lists L = $\{(a_1,b_1), (a_2,b_2),....(a_s,b_s)\}$ and R = $\{(x_1,y_1),(x_2,y_2),....(x_t,y_t)\}$.

Output: A sorted list H = $\{(c_1,d_1), (c_2,d_2),....(c_u,d_u)\}$ where $u \leq s + t - 1$.

Steps:

1. Initialize H:=$\emptyset$, $i = 1$, $j = 1$, $k = 1$.
2. Calculate (c_k, d_k) as, $(c_k, d_k) = (a_i + x_j, max(b_i, y_j))$.
3. Update H as, $H = H \cup (c_k, d_k)$.
4. Increase the k value by 1.
5. If $max(b_i, y_j) = b_i$, then increase i value by 1.
   If $max(b_i, y_j) = y_j$, then increase j value by 1.
6. If $i \leq s$ and $j \leq t$, then repeat steps $2 - 5$.

Similarly an algorithm called *horizontal node sizing* is designed, when the clustering at a node is in vertical direction. In this algorithm, $(c_k, d_k) = (max(a_i, x_j), b_i + y_j)$ and b and y in step-5 to be replaced by a and x respectively.

These algorithms work in bottom-up fashion until it reaches the root node. When two modules are clustering, either horizontal or vertical node sizing has to be done. Thus, at each clustered node, number of possible implementations (size of the list) increases. At the root node, all pairs in the list are examined, and one achieves minimum area is selected. Thus this algorithm gives an optimal area implementation of the sliceable floorplan.

Consider again the example shown in figure 3.4. Let, many possible implementations are available for modules a, b, c, d, e. According to the connectivity graph, first a and c are combining in vertical fashion. Horizontal node sizing algorithm to be performed to get all possible sizes for node ac. Similarly do horizontal node sizing algorithm for b and d to get all possible sizes for node bd. Now perform vertical node sizing with ac and bd, since we are combining these nodes horizontally. This process to be continued until reaches the root node and the area of the floorplan can be calculated from all possible implementations available at the root node.

A sample result of the optimal area implementation algorithm, generated from a connected graph has been attached in section 10.1.

Chapter 4

Placement

The goal of placement is to find the best position for each module on the chip according to the appropriate cost functions. The shape and terminal location of modules and netlist are the input to the process. Placement algorithms can be divided into two major classes, iterative improvement and constructive placement. Iterative improvement starts with an initial placement and then it is evaluated and repeatedly modified for a better solution, in which the cost is reduced. In constructive placement, a good placement is constructed in a global sense.

4.1 Placement by Simulated Annealing

Simulated annealing is an important algorithm in the class of iterative, probabilistic algorithms. It is based on the thermodynamic processes for growing crystals. By the annealing process, solid state material can get very close to perfect crystal, which represents a configuration with a global minimum amount of energy. In this process, the material is melted, and then cooled very slowly, according to a specific schedule of decreasing the temperature. If the initial temperature is high enough to ensure a sufficiently random state, and if the cooling is slow enough to ensure that thermal equilibrium is reached at each temperature, then the atoms will arrange themselves in the form of a perfect crystal.

In the simulated annealing, the energy is replaced by the cost function, which we want to minimize. Also, considers the modules to be placed as the atoms, and thus the feasible solutions replaces the states of matter. It starts with a random initial placement, called current solution, at high temperature. A new placement is generated from the current one. Change in cost ΔC, where $\Delta C = New\ solution\ cost - Current\ solution\ cost$, is calculated and if $\Delta C < 0$, new solution replaces the current one. If $\Delta C \geq 0$, to avoid being trapped in a local minimum, new solution is accepted with some probability. Probability of accepting a bad solution is high at high temperature. As the algorithm proceeds, the temperature decreases, and such movement is less likely to be accepted. When the algorithm terminates, the current solution gives the best among all the solutions found.

The quality of the solution generated by the simulated annealing algorithm depends on the initial value of temperature and the cooling schedule. Basically two ways of choosing the next temperature, a static and a dynamic one. In static $\mathrm{T}_{next} = \alpha\mathrm{T}$, where $0 < \alpha < 1$ and α is close to 1. In dynamic, the choice of the next temperature is determined using the standard deviation of the cost distribution at the current temperature.

The simulated annealing algorithm can be implemented by **Scilab**. To use the Simulated annealing(SA) algorithm in Scilab, the following steps to be performed.

- Build the neighbouring function, the acceptance function, and the temperature law function.

- Configure the list of parameters which includes the neighbouring, acceptance and temperature functions in addition to the normal parameters, by using *init_param* and *add_param*.
- Compute an initial temperature by calling an SA function, *compute_initial_temp*.
- Find an optimum by using the *optim_sa* solver.

The *compute_initial_temp* function computes an initial temperature, given an initial probability of accepting a bad solution. This computation is based on some iterations of random walk through the solutions and their evaluations by the cost function(objective function). Thus the arguments of this function are initial solution, cost function, initial probability of accepting a bad solution, number of random walk and the list of parameters.

The neighbouring function, computes a neighbour of a given point or the next solution from current solution. Prototype of the function to be build is `function` *next_soln = neighbour_fn(current_soln, T , param)* where *current_soln* and *next_soln* represents the current and next solution respectively, T represents current temperature, and *param* represents list of parameters.

The acceptance function decides whether SA uses the fast simulated annealing process or exponential based process, where the default one is the exponential based one.

The temperature law function, updates the temperature while the iterations are processed. The default temperature law uses the static method where, $\mathrm{T}_{next} = \alpha\mathrm{T}$.

The *optim_sa* function implements the simulated annealing solver to find the best solution. It is based on the iterative update of the current solution, by taking into account the neighbouring function and the acceptance function. The solution which achieves the minimum of the objective function over the iterations is the best one. This function is based on the following steps.

1. Calculate initial solution and initial temperature.
2. Initialize *count*, as *zero*.
3. Compute a neighbour of the current solution.
4. Compute the cost for that neighbour.
5. If the objective decreases or if the acceptance criteria is true, then overwrite the current solution with the neighbour.
6. If the cost of the best solution is greater than that of the current solution, overwrite the best cost with the current cost.
7. Increase the count by 1.
8. If the count is less than some specific value repeat the steps $3 - 7$.
9. Update the temperature with the temperature law.
10. Repeat the steps $2 - 9$, if the algorithm not reached the termination condition.

Thus the arguments to the *optim_sa* function are, initial solution, cost function, number of iterations in the external loop and internal loop, initial temperature, and parameter list.

4.1.1 Standard Cell Placement by SA:

In standard cell design, all modules have the same height so that they can be arranged in rows. The inputs to the placement problem are module descriptions and the netlist. The output is the x and y coordinates of the modules. The main goal in this problem is to reduce the cost function which

can be based on the wire length of each net. The wire length of a net can be calculated by half perimeter method. In this method, half the perimeter of the smallest bounding box enclosing the net is the required wire length.

Major steps for solving SA with the help of *Scilab*, are the construction of initial solution, initial temperature, neighbouring function to move from one solution to another and the cost function.

Consider an example of 8 modules, $\{A, B, C, D, E, F, G, H\}$, to be placed in two rows, four modules in each row. Let the given positions be $\{1, 2, 3, 4, 5, 6, 7, 8\}$. Arrangement of these positions is shown in figure 4.1(a). Any one feasible solution can be selected as the initial solution. Let the initial solution be such that module A in position 1, B in position 2 etc. Module and its associated position number for initial solution is shown in figure 4.1(b). The dimension of the positions will be same as that of the modules associated to that. Thus the dimension of position 1 and 2 will be equal to that of A and B respectively. Interchange of two modules will give the next solution, as shown in figure 4.1(c). Now module B in position 1 and A in position 2. By continuing this interchange, we can move through all feasible solutions.

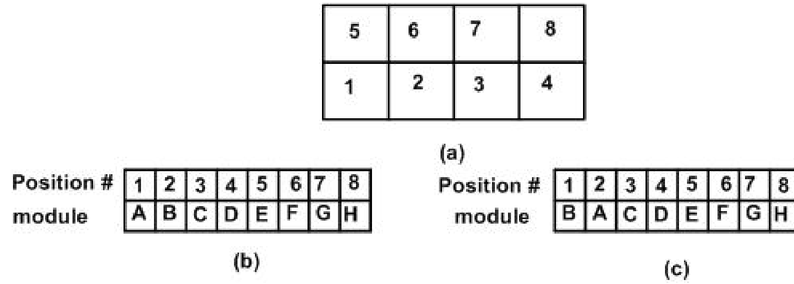

Figure 4.1: (a) Arrangement of positions (b) Module and associated position number for initial solution (c) New solution by interchanging first two modules.

We can calculate the coordinates of all modules from their positions in the solution. In each row,

$$x_{pi} = \sum_{n=1}^{i-1} W(M_{pn}) \qquad when \quad i \neq 1, \tag{4.1}$$

where x_{pi} is the x coordinate of module at *position i* and *W(M_{pi})* is the *width of module at position i.* and

$$x_{pi} = 0 \qquad when \quad i = 1 \tag{4.2}$$

Similarly y coordinates can be calculated from the height of the module. Based on these coordinates we can calculate the length of each net and thus the cost. Here the cost is calculated by the formula,

$$cost = \sum_{eachnet} (connectivity * wirelength). \tag{4.3}$$

Initial temperature can be calculated by calling the function *compute_initial_temp* which uses the initial solution, cost function and the parameter list as the arguments. The parameter list includes inputs and the neighbouring function. With these functions and values as arguments, we can call *optim_sa* function to calculate the coordinates of the modules corresponding to the minimum cost.

A sample result of the standard cell placement with the help of scilab has been attached in section 10.2.

4.2 Placement by Genetic Algorithm

The objective of the Genetic algorithm (GA), based on the natural process of evolution, is to find the optimal solution to a problem. But since it is a heuristic procedure, not guaranteed to find the

optimum, but will find a very good solution for a wide range of problems.

GA works on a collection of several alternative solutions called *population*. Each solution or individual in the population is called *chromosome* and individual character in this is called *genes*, which can be the position of one module in a placement problem. To obtain better solutions (population) from existing one, a new *generation* is evolved in each iteration of the GA.

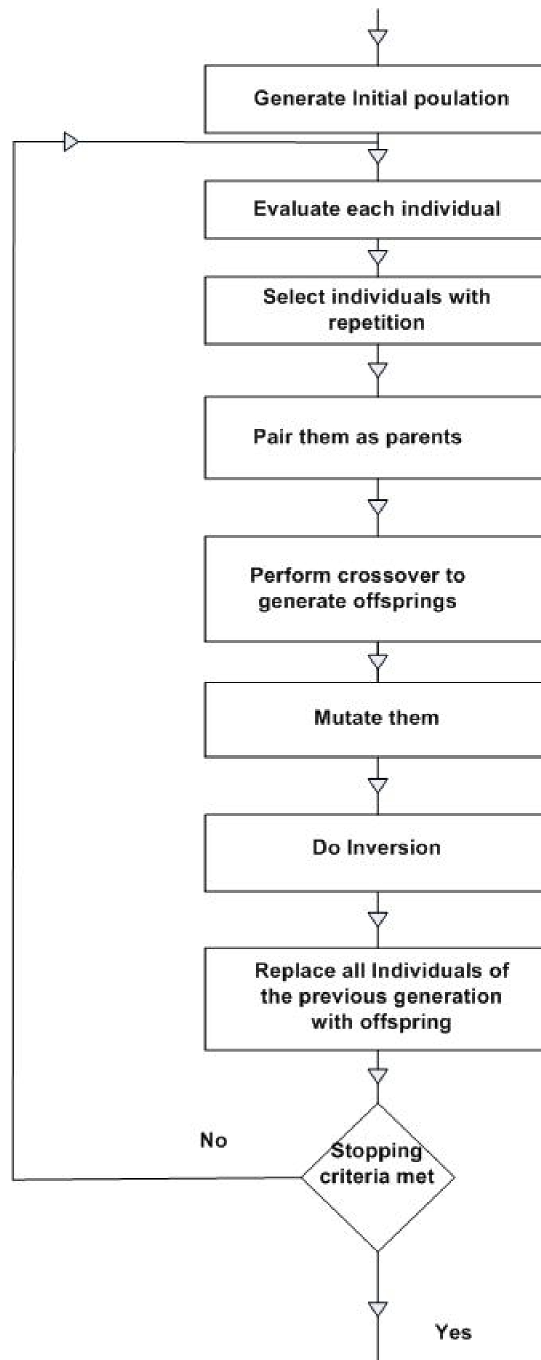

Figure 4.2: Flow chart of simple Genetic Algorithm

The *generation gap* is the fraction of individuals in the population that are replaced from one generation to the next. Based on this, there are two basic GA approaches, called *simple GA* and *steady state algorithm*. Generation gap is equal to 1 in the *simple GA* and is less than 1 for the other approach.

Generation of a new population involves various steps. First evaluate each individual of the population by a user defined *fitness* function, which is opposite to the cost function. Then highly fit individuals are selected from the population for reproduction. Selected individuals form pairs called *parents*. Different operations for reproduction are *crossover*, *mutation* and *inversion*.

In the *crossover* operation, portions of two parents are combined to produce two new individuals, called *offsprings*. For each pair of parents, crossover is performed with a *crossover probability*, P_c. New features can be introduced into a population by *mutation*. It produces random changes in the offspring with a probability called *mutation probability,* P_M. Crossover is the main operation to search the solution space, but does not guarantee the reachability of the entire solution space with a finite population size. Mutation improves search space by introducing new genes into the population.

Inversion does not change the solution represented by the chromosome , but changes the sequence of genes in it. It generally improves the efficiency of the crossover operator. If the genes are arranged in the chromosome such that the related genes are close together from the beginning, then inversion is not required. *Inversion probability* is denoted by P_I.

The flow chart of the simple genetic algorithm is shown in figure 4.2. Here, the initial generation can be random or user specified. After the reproduction, new generation will replace the old one and evolve until some stopping criterion is met.

4.2.1 Macro Cell Placement by GA:

Macro cell placement problem, considers a set of rectangular cells with different width and height. The objective is to place these blocks in such a way that the total wire length and the chip area are minimized. The wire length can be calculated by the half perimeter method. Input to the problem is the module descriptions and the netlist. Output is the list of x and y coordinates and the orientation of these modules. Like standard cell placement each position is represented by a position number, but they are not with the same height and orientation.

Consider an example of 8 modules with different dimensions and orientations, arranged in two rows. First construct an initial population randomly. Two solutions in a population are given with module number and its orientation in figure 4.3 (b) and (c). If the position or orientation of any module is different, then the solution is also different.

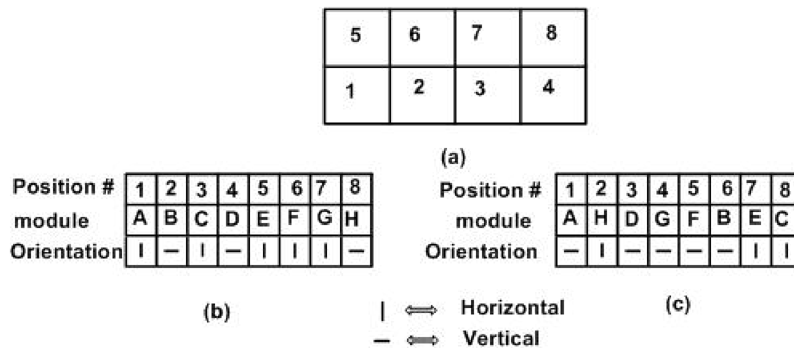

Figure 4.3: (a) Arrangement of positions. (b)and(c) Two solutions in the initial population.

Evaluate all solutions in the population by a fitness function which can be considered as the inverse of the cost function. Sort the individuals according to the fitness values. Select the solutions for reproduction based upon their rank in the sorted list. Repetition of best solutions will force the rejection of worst to keep the population size same. Assume figure 4.3 (b) and (c) solutions are selected for reproduction.

Choose any random value between 1 and 8 and then do one point crossover operation. Copy

the position, module and orientation values on the left of the cut point, from one parent to one offspring, and from second parent to second offspring. Fill the right portion of the first offspring by going through the second parent, from the beginning to the end and taking those elements which were left out, in order. Similarly fill the second offspring from the parent one. An example is shown in figure 4.4.

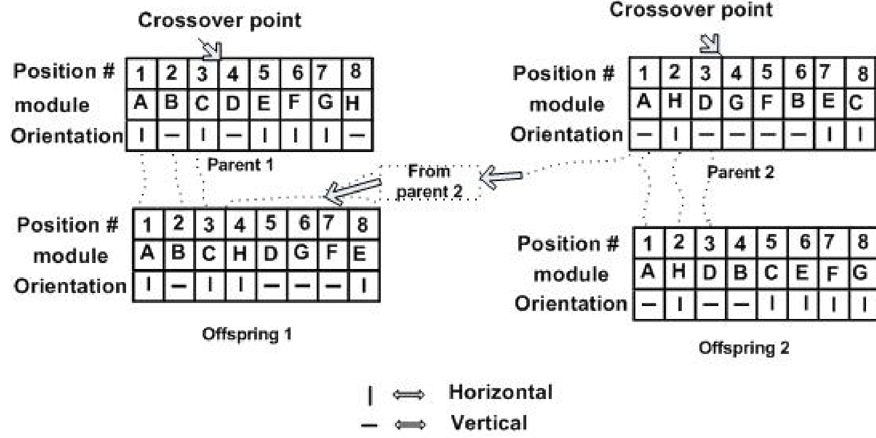

Figure 4.4: Crossover at a random point to generate two offsprings from two parents

Now the mutation can be done in many ways. One method consists of choosing a random value and changing only the orientation of the cell at that point. This causes a minor change in the solution. After these reproduction methods, replace the entire population with the offsprings. Now continue the process of evaluation, reproduction etc, for a fixed number of generations to get a better solution. Similar to standard cell placement, x and y coordinates of the solution can be calculated from the position of the modules. Based on these coordinates, we can calculate the half perimeter length of each net and thus the cost and the fitness values of each solution.

A sample result of the macro cell placement by genetic algorithm, with 12 initial solutions, has been attached in section 10.3.

4.3 Regular Placement: Assignment Problem

In this problem, possible positions of the modules, called target cells are predetermined. Each module to be assigned to a target. This placement problem has two parts called relaxed placement and removing overlaps. The linear programming method is applied to the relaxed placement problem and removing overlapping can be done by an assignment problem. Here some modules have preassigned locations. Overlapping of modules is allowed in the first step.

For each net N_i, three variables are defined.

X_{li} : The left most x coordinate of net N_i,
X_{ri} : The right most x coordinate of net N_i, and
X_{vi} : Possible x coordinate of module M_v, where module M_v is connected to net N_i.

But according to definition,

$$X_l \leq X_{li} \leq X_{vi} \leq X_{ri} \leq X_r, \tag{4.4}$$

where X_l and X_r are the values of the left and right boundaries of the chip, respectively.
For fixed modules,

$$X_{vi} = X_i. \tag{4.5}$$

where X_i is the predetermined value.
Now the objective is to minimize the cost function defined by,

$$\sum_{N_i \epsilon N} W(i)(X_{ri} - X_{li}) \tag{4.6}$$

where $W(i)$ is the weight of each net N_i.
Similarly, three equations can be constructed for the y coordinate also. Linear programming problem can be formulated by using all these six equations.

These equations can be solved by using *LP solver*. Static library of *lp solve* can be linked with the application, like *C*. After linking with the application, the *lp solve* functions can be called, like they are linked to the program, by including the header files. Based on the equations, 4.4 and 4.5, inequality and equality constraints can be constructed in LP solver, by the function *add_ constraintex()*. These constraints are added one at a time and to be arranged in different rows. Equation 4.6 is used to construct objective function by using *set_ obj_fnex()* and *set_ minim()*, and then solve by the function *solve()*.

These solutions may have overlapping modules, which should be removed. Assign a cost C_{ij} to each module M_i if it is assigned to a target H_j. Typically, the cost C_{ij} is the distance between the current position of M_i and the target H_j. Now, the modules should be placed in such a way that no overlapping exists between the modules and the total cost is minimum. Since we have one module to one target, this becomes a job assignment problem. A sample result of the implementation has been attached in section 10.4. The algorithm can be summarized as follows:

Input: Netlist, Fixed module locations, Boundaries of the chip and the Target locations.

Steps:

1. Generate equations for the variables of each net and module.
2. Convert these equations to constraints and objective function and solve by LP solver to obtain a relaxed placement solution.
3. Calculate the cost C_{ij}, which is the distance between the module M_i and the target H_j, for each module and target.
4. Use a job assignment problem to assign the modules to the targets with minimum cost.

4.4 Module Placement Based on Resistive Network

In this method an electrical analogy is considered, where module position is analogous to node voltage. Because of network analogy, the cost function used is the sum of the squared wire lengths. Let there be n modules placed at positions (x_i, y_i), $i = 1, 2, ..., n$. Let c_{ij} denotes the connectivity between module i and j. The cost function is given by

$$\Phi(x, y) = \frac{1}{2} \sum_{i,j=1}^{n} c_{ij}[(x_i - x_j)^2 + (y_i - y_j)^2] = x^T Bx + y^T By, \tag{4.7}$$

where

$$B = D - C \tag{4.8}$$

is an $n \times n$ symmetric matrix, $C = [c_{ij}]$ is the connectivity matrix and D is a diagonal matrix whose i^{th} element $d_{ii} = \sum_{j=1}^{n} c_{ij}$. With the symmetry between x and y in equation 4.7, only one dimensional problem to be considered for optimization.

4.4.1 Network Analogy

We can model x_i, the x coordinate of module i, with a node voltage v_i at node i. B in equation 4.8 is similar to the indefinite admittance matrix, Y_n, of a n-terminal linear passive resistive network, shown in figure 4.5, where c_{ij} is the mutual admittance between node i and j and d_{ii} is the self admittance at node i[14]. Since the power dissipation in the resistive network is given by,

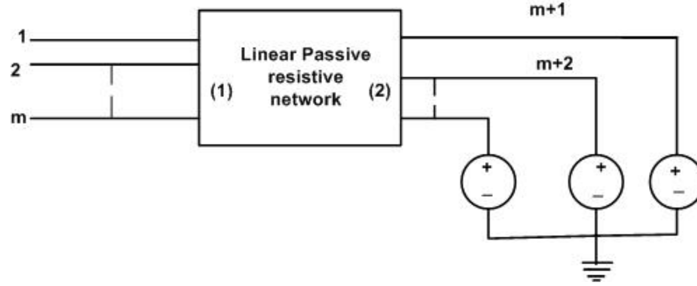

Figure 4.5: An n terminal passive resistive network where first m nodes are floating

$$P = v^T Y_n v, \tag{4.9}$$

the cost function of the placement problem becomes the power dissipation in the network, in which the solution is always in such a way that power dissipation is minimum[14]. Let the voltage vector v is represented by

$$v = \begin{bmatrix} v1 \\ v2 \end{bmatrix}$$

where the node voltages corresponding to fixed modules are v_2, and that of the movable modules are v_1. The network equations are,

$$0 = y_{11}v_1 + y_{12}v_2, \tag{4.10}$$

$$0 = y_{21}v_1 + y_{22}v_2. \tag{4.11}$$

Equation 4.10 gives,

$$v_1 = -y_{11}^{-1} y_{12} v_2, \tag{4.12}$$

which gives the solution of the movable modules in terms of the fixed one.
But, since the modules should be placed on the given slots, voltage vector v_1 must represent a set of legal values specified by p. Let $p = [p_1, p_2, \dots\ p_m\ ]^T$, and $v_1 = [x_1, x_2, \dots\ x_m\ ]^T$, where p_i is the legal value and x_i the actual voltage at node i and m is the number of movable modules. The following set of equations represents the slot constraints,

$$\begin{aligned} \sum_{i=1}^{m} x_i &= \sum_{i=1}^{m} p_i \\ \sum_{i=1}^{m} x_i^2 &= \sum_{i=1}^{m} p_i^2 \\ &\cdots \\ &\cdots \\ \sum_{i=1}^{m} x_i^m &= \sum_{i=1}^{m} p_i^m \end{aligned} \tag{4.13}$$

The first equation can be rewritten as

$$I^T v_1 = I^T p \equiv d \tag{4.14}$$

where d is the sum of the legal values. From equations 4.9, 4.10, and 4.11, now the problem becomes the minimization of

$$P = v^T Y_n v = v_1^T y_{11} v_1 + 2v_1^T y_{12} v_2 + v_2^T y_{22} v_2 \tag{4.15}$$

subject to the set of constraints, in equation 4.13. Since designing an exact algorithm for solving this, is not practical, the following heuristics is proposed[14].

4.4.2 Proposed Method

In this method the problem is divided into subproblems of optimization, scaling, relaxation, and partitioning[14]. Node voltages and module coordinates are interchangeable according to the context.

1. **Optimization:** In this stage, the above problem is solved only by considering the linear constraint in equation 4.13. Then the solution is given by the Kuhn-Tucker conditions as[14],

$$v_1 = -y_{11}^{-1}[y_{12}v_2 + i_1], \tag{4.16}$$

where

$$i_1 = \frac{d + I^T y_{11}^{-1} y_{12} v_2}{I^T y_{11}^{-1} I} I, \tag{4.17}$$

But since only linear constraint is considered, the solution will not put modules on slots, but concentrated at the center of gravity of the modules. Now these modules to be distributed, which leads to the increment of power dissipation. The increase in power dissipation has an upper bound proportional to δv^2. Thus the problem is now to minimize the increase in power, which is called as *scaling*.

2. **Scaling:** Here the objective is to minimize the increase in power dissipation subject to the linear and second order (first two) constrains in equation 4.13.

Let us assume that, there are k modules in the region where distribution of modules to be done. Let $[p_1,p_2, \ldots\ p_k\ ]^T$ be the legal values, $[x_{o1},x_{o2}, \ldots\ x_{ok}\ ]^T$ be the initial optimal solutions and let the new solutions after scaling are given by $[x_{n1},x_{n2}, \ldots\ x_{nk}\ ]^T$. Thus the problem is to minimize,

$$\sum_{i=1}^{k} (x_{ni} - x_{oi})^2 \tag{4.18}$$

under the constraints

$$\sum_{i=1}^{k} x_{ni} = \sum_{i=1}^{k} p_i \tag{4.19}$$

and

$$\sum_{i=1}^{k} x_{ni}^2 = \sum_{i=1}^{k} p_i^2 \tag{4.20}$$

This leads to the solution[14],

$$x_{ni} = \frac{x_{oi} - c_o}{a_o} a_n + c_n \tag{4.21}$$

where

$$c_n = \frac{1}{k} \sum_{i=1}^{k} p_i \tag{4.22}$$

$$a_n = \sqrt{\frac{1}{k}\sum_{i=1}^{k}(p_i - c_n)^2} \quad (4.23)$$

$$c_o = \frac{1}{k}\sum_{i=1}^{k} x_{oi} \quad (4.24)$$

and

$$a_o = \sqrt{\frac{1}{k}\sum_{i=1}^{k}(x_{oi} - c_o)^2} \quad (4.25)$$

Now for further improvement relaxation will be done.

3. **Relaxation:** In this stage, repeated use of scaling and optimization will be performed in the region specified by the user. The steps to be followed are,
   (a) Collect the initial values from the optimization stage and arrange them from left to right according to coordinates.
   (b) Do the scaling in the region enclosing some (as per the designer) values from the left while considering the remaining as fixed.
   (c) Now fix these scaled (new) values, and release the remaining movable modules in the right, for optimization.
   (d) Now the last two steps repeat by considering first right region values and then the values in the middle region.
4. **Partitioning:** Now divide the region into two. Once again do the scaling for the left part and then the right part. The result of this gives two partitioned subregions together with their associated modules. Now repeat the process of optimization, scaling, relaxation and partitioning separately for each subregion. Finally the modules will be assigned to the legal values.

4.5 Timing Driven Placement

It incorporates timing objective functions into the placement problem. If a placement is bad in terms of timing, even a very powerful router cannot help[1]. One of the approaches for the timing driven placement problem is *zero slack algorithm.*

4.5.1 Zero Slack Algorithm

Consider a circuit without any loops, inputs to the circuit are called as *primary inputs* (PIs) and outputs are called as *primary outputs*(POs). Assume, the signal arriving time at each PI, signal required time at each PO and the Gate delays, are given. The *arrival time* t_A of the signal at inputs and outputs of each gate (each pin) can be calculated by tracing the signal path starting from the PIs. Similarly the *required time* t_R at each pin is obtained by starting from the POs and tracing back the path of the signal. If a gate has k distinct output signals, that gate will be duplicated k times, with the same input and distinct output.

Calculation of arrival time: Arrival time at the gates are calculated by visiting the gate one by one. A gate can be visited, if either its inputs are PIs or all the gates connected to its inputs are

visited already. Let t_A^j be the arrival time of an output pin j, and t_A^1, t_A^2,....,t_A^m are that of the output pins of its fan in cells. The relationship between them is expressed as[1],

$$t_A^j = \max_{i=1}^{m}(t_A^i + d_{n_i}) + D \tag{4.26}$$

where D is the delay of the cell containing the pin j, and d_{n_i} is the delay of the net i which connects the cells. This relation is illustrated in the Figure 4.6 (a) .

Calculation of required time: Let t_R^j be the required time of an output pin j, and t_R^1, t_R^2,....,t_R^m are that of the output pins of its fanout cells. The relationship between them is shown in figure 4.6 (b), and is expressed as[1],

$$t_R^j = \min_{i=1}^{m}(t_R^i - D_i) - d_n \tag{4.27}$$

where D_1, D_2,..,D_m are cell delays, and d_n is the delay of the net that connects the cells to its fanout cells. Also define

$$slack = t_R - t_A. \tag{4.28}$$

The objective of the zero slack algorithm is to associate maximum allowable delays to each net.

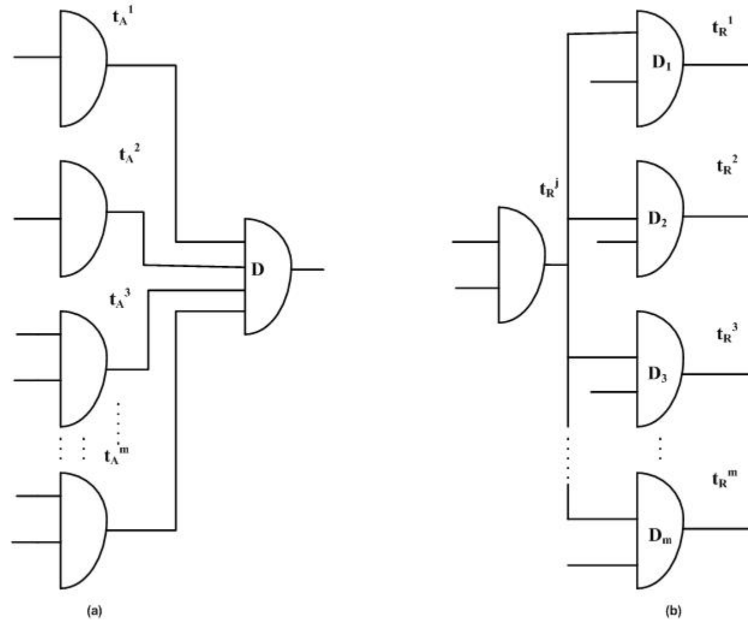

Figure 4.6: (a) Computation of arrival time at an input pin (b) Computation of required time at an output pin

Thus it converts timing constraints to net length(delay) constraints. It works in an iterative manner with initial delay of each net as zero, finally maximum possible delays. This results zero slacks for primary inputs, outputs and the output of each cell.

Figures 4.7-4.9 demonstrates an example which uses zero slack algorithm. Arrival time of primary inputs and required time of primary outputs are given. Consider the initial delay of the net as zero. Now slacks can be calculated at each pin according to the equations 4.26 - 4.28. A triplet t_A/ slack/t_R is shown in the figure 4.7, at each pin . Calculate the minimum positive slack, here it is 4, and then select its corresponding path segment. Circled triplets indicates these path. Now distribute these four units of slack along the path segment. Result is shown in figure 4.8. It indicates that each of the nets on the path can have length (delay) proportional to the assigned length in the layout. Now slacks on this path segment becomes zero. Then the new slacks are calculated and repeated this steps until no positive slack exists. Now the slack at the output pins of each gate is

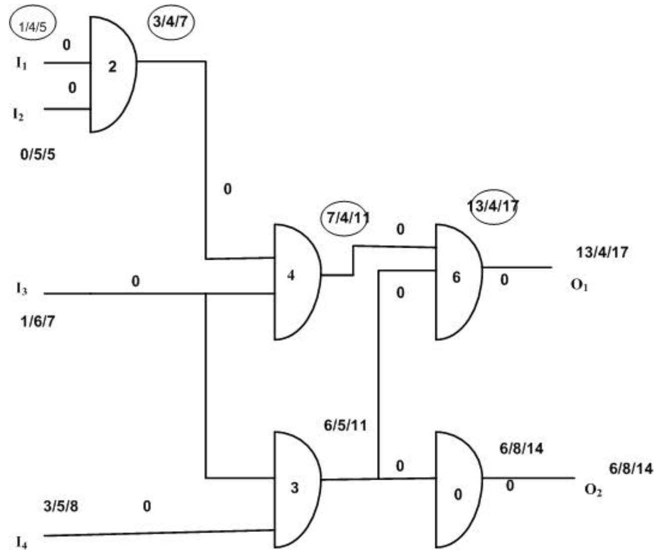

Figure 4.7: Zero slack algorithm example, Step:1

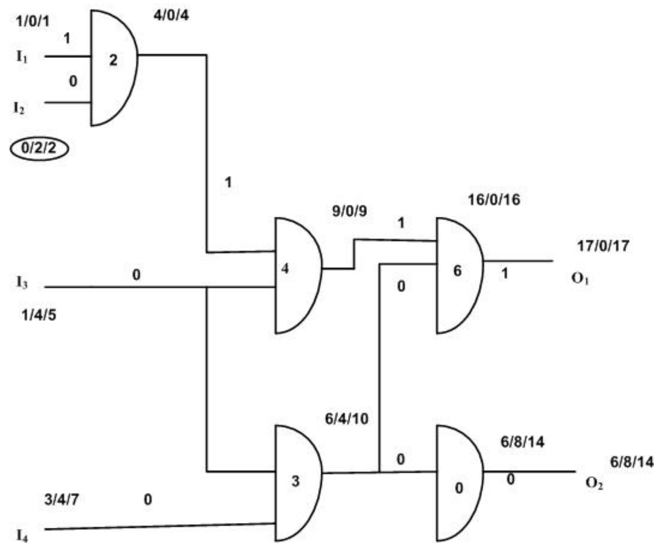

Figure 4.8: Zero slack algorithm example, Step:2

zero as shown in figure 4.9.

A sample result of the implementation of this algorithm has been attached in section 10.5. The algorithm can be summarized as follows,

Input: Arrival time of each input, Required time of each output and Delay of each gate.

Steps:

1. Assume the delay of each net as zero.
2. Calculate the arrival time, required time and slack time of each gate, input and output.
3. Calculate the minimum positive slack.
4. Calculate a path with all the slacks equal to the minimum positive slack.

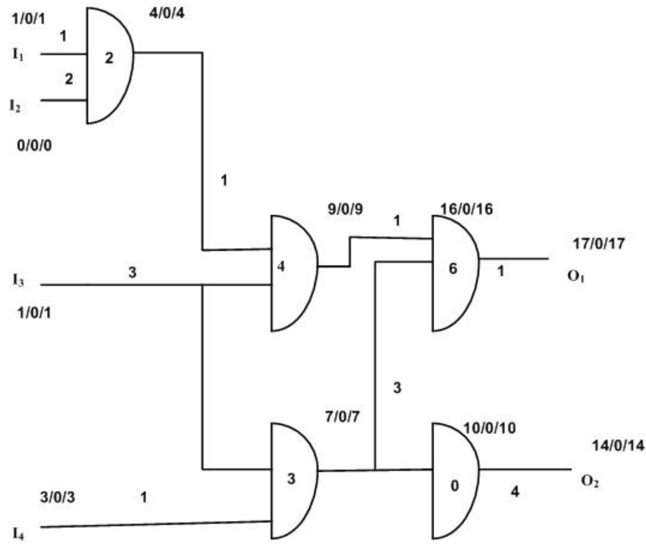

Figure 4.9: Zero slack algorithm example, Last Step

5. Distribute minimum slack value along the path segment. (now, slacks on the minimum path segment becomes zero.)
6. Calculate the new slacks.
7. Repeat steps $2-6$, until no positive slack exists.

After the computation of the maximum allowable net delay, any placement algorithm can be used to produce, a timing-driven placement. Actually weights on the nets are inversely proportional to the delay calculated. Modules connected with nets of small possible delay value, to be placed close to each other. Such path with small maximum allowable delay value are called *critical path*[1]. This algorithm have been used widely to generate timing constrains for nets.

Chapter 5

Routing

The objective of the routing problem is to determine the geometrical layout of all signal nets while minimizing the chip area and total wire length. These signal nets must be routed within the spaces reserved during the placement process. Typically, it is carried out in two steps, global routing and detailed routing.

5.1 Fundamentals

Some of the fundamental concepts involved withe these routing problems are maze running and steiner trees.

5.1.1 Maze Running

It is an algorithm to find a shortest path between two points on a grid with obstacles. Also called as Lee's algorithm or the Lee-Moore algorithm. The steps in this algorithm are,

1. Start from any cell, called as *source cell*, and label the cell as 0.
2. Label all unblocked cells adjacent to the source as 1. The unlabeled and unblocked cells, adjacent to these should be labeled as 2. Two points are called adjacent only if they are either horizontally or vertically adjacent.
3. In general, label all the cells, which are unlabeled and unblocked, adjacent to the label n, as *n+1*.
4. Repeat these labeling till the *target cell* is reached.
5. Now trace back from the target to the source, by following the labels in the descending order. If there are more than one such path, choose any one.

The final labeling and one of the shortest path is shown in Figure 5.1. A major drawback of this algorithm is the requirement of large amount of memory to label the grids[3]. There are many attempts to reduce this difficulty. One solution is to use a scheme where a grid just pointing to its neighbours instead of storing the value. Another effective method, called bidirectional search, is to do the searching or labeling from both the source and the target till it meets.

5.1.2 Steiner Trees

A general steiner minimum tree(SMT) problem is defined as follows: Given a graph $G(V, E)$ with edge weights and a set of *required* vertices $S \subseteq V$, find a tree of minimum cost that spans S. Such

		8	7	Target -8				
		7	6	7	8			
8	7	6	5	6	7	8		
7	6		4	5	6	7	8	
6	5	4	3	4	5	6	7	8
5	4	3	2	3		5	6	7
4	3	2	1	2	3	4	5	6
5		1	0 Source	1	2	3		7
4	3	2	1	2	3	4	5	6

Represents blocked cells

A shortest path from source to target is shown by the line

Figure 5.1: Labeling by the Lee-Moore algorithm with one shortest path

a tree is permitted to touch other nodes, called *steiner points*. Important steiner tree problems are, *Minimum length steiner tree* in which the goal is to minimize the sum of the length of the edges of the tree, and *Weighted steiner tree* in which the goal is to minimize the total cost of the tree. In weighted steiner tree, a plane is partitioned into a collection of weighted regions. Cost of an edge with length l, in a region of weight w, is lw. In a routing problem, the objective is to minimize the length and traffic through the routing areas. These can be modeled using weighted steiner trees.

In SMT, if $\mid S \mid = 2$, the problem is similar to the shortest path problem, which is simple. Another similar, but easy problem is minimum spanning tree problem, where $\mid S \mid = V$. But in general, SMT problem is hard. Since it is hard, most of the steiner tree algorithms, aim at finding a good steiner tree by doing a mixture of minimum spanning tree and shortest path[12]. The approximation algorithm uses the following heuristics strategy[4]:

1. Given $G(V, E)$ and *required* vertices $S \subseteq V$, graph $G_c(S, E_c)$, can be constructed as follows:
   (a) Vertices, S, of G_c is same as the required vertices of G.
   (b) G_c is a complete graph with edges, E_c, between all pair of vertices.

(c) From the graph G, calculate the shortest path lengths between each pair of required vertices. Weight of the edges E_c, represents the corresponding shortest path lengths.

An example of G with required vertices $\{a, b, c, d, e\}$ and its corresponding G_c are shown in figure 5.2 (a) & (b) respectively.

2. Find T_c, which is a minimum spanning tree of G_c, as shown in figure 5.2(c).

3. Construct G_{sub}, which is a subspace of G, by replacing each edge of T_c by the corresponding edges in G. An example is given in figure 5.2(d), where edge $a - d$ of T_c is replaced by the corresponding edges $a - f - i - d$ of G. Similarly all the edges of T_c to be replaced.

4. Find a minimum spanning tree, say G_1, of G_{sub}. If any cyclic path exists in G_{sub}, that will be removed by this step.

5. Delete leaves in G_1, which are not the required vertices. This step is called as *pruning*.

The steiner tree construction is illustrated in Figure 5.2 with input graph in figure 5.2(a) and final steiner tree with required vertices $\{a, b, c, d, e\}$ and steiner terminals, $\{f, i\}$ is in figure 5.2(e).

A sample result of the construction of steiner tree from an input graph has been attached in section 10.6. This approximation algorithm yields steiner tree that is not worse than twice the

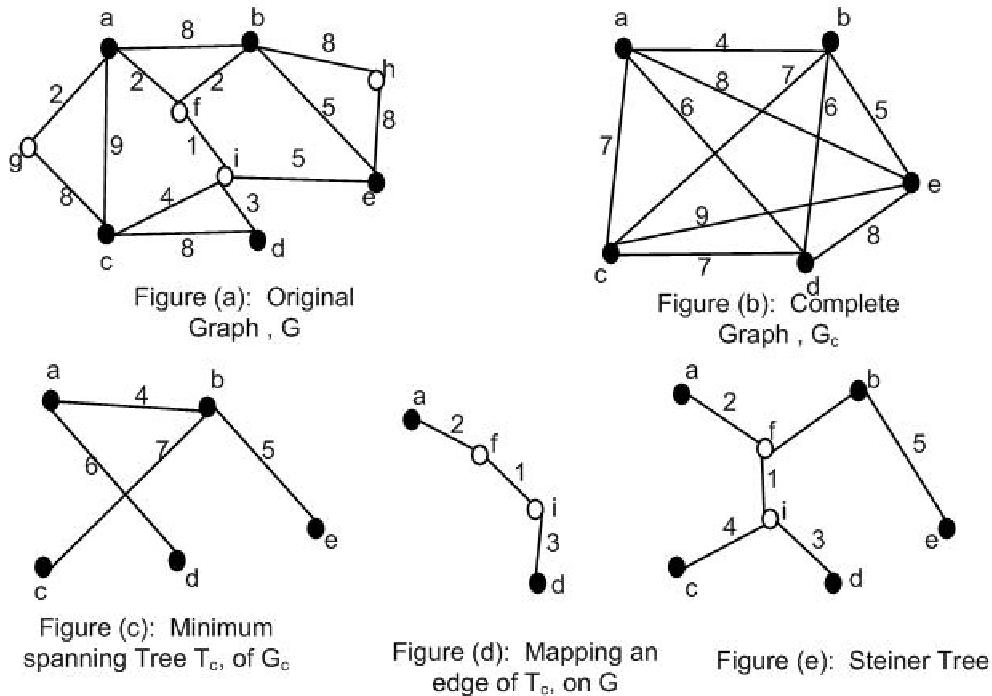

Figure 5.2: Construction of Steiner tree

optimal steiner tree.

5.2 Global Routing

A global router decomposes a large routing problem into a small and manageable subproblems. This decomposition is carried out by finding an approximate path for each net, which will reduce the chip size, shorten the wire length and reduces the congestion. There are several approaches to global routing, one of them is *randomized routing*, which uses *linear integer programming method.*

5.2.1 Randomized Routing

Objective of this problem is to select a suitable steiner tree for each net. Let n_i be the number of steiner trees possible for the net i, and let these trees be $[T_i^1, T_i^2, ..., T_i^{n_i}]$.
Define variables, x_{ij}, for each net i and tree T_i^j such that, $x_{ij} = 1$ if net i uses the tree T_i^j, otherwise, $x_{ij} = 0$.

Let $U(e)$ is the traffic in the edge e. Calculation of $U(e)$ depends upon the selected trees which contain the edge e. If a particular tree, containing e, is selected for the routing of a net, then the cost associated with that net to be considered in the calculation. This computing we have to do with all the trees.

Let $f(e)$ indicates whether the edge e belongs to a particular tree or not. Variable $f_{ij}(e)$ is defined for each net i and tree T_i^j such that,

$$\begin{aligned} f_{ij}(e) &= 1 \;\; if\ edge\ e\ \in\ T_i^j\ of\ net\ i \\ &= 0, \qquad\qquad\qquad otherwise \end{aligned}$$

Thus the equation for the calculation of $U(e)$ is,

$$U(e) = \sum_{j=1}^{n_i} \sum_{i} w(i) f_{ij}(e) x_{ij} \tag{5.1}$$

where $w(i)$ is the cost factor for the net i.

The capacity constraint of the edge e, $U(e) \ \le\ c(e)$, where $c(e)$ is the maximum capacity of the edge e, is eliminated in this version. It ensures that, all nets are routed by any one possible tree. So, for each net i,

$$\sum_{j=1}^{n_i} x_{ij} = 1 \tag{5.2}$$

Instead of capacity constraint, the new equation to be considered is,

$$U(e)/c(e) \le x_L \tag{5.3}$$

where, x_L denote the maximum load on any edge. The objective is to minimize the cost function defined as,

$$c = \lambda x_L + \sum_{e \in E} l(e) U(e) \tag{5.4}$$

where, λ is the penalty for exceeding the capacity of edge, $l(e)$ is the length of the edge e, and E is the set of edges in the routing graph.

By Equations 5.2, 5.3 and 5.4, the problem is formulated as linear programming, and *LP solver* can be used to find a solution for x_{ij}. While using the *LP solver*, first the total number of variables, here all x_{ij} and x_L, to be defined as the number of columns. Start building the model row by row by setting the function *set_add_rowmode*(_, *TRUE*) and then add the constraints one by one by the function *add_constraintex*(). After completing the construction of all the constraints, reset the row mode by *set_add_rowmode*(_, *FALSE*), then build the objective function by *set_obj_fnex*(). The functions *set_minim*() and *solve*() can be used for calculating x_{ij} and x_L, which minimizes the objective function with the constraints.

The idea used in randomized routing is to get an initial solution, by omitting the integral constraint[13]. In the next step, get an integer solution close to the optimal solution, where the lower bound on cost is by the initial solution. The algorithm can be summarized as,

Input: All possible steiner trees for each net. Also the cost factor of each net and length and

maximum capacity of each edge.

Steps:

1. For each edge e, calculate $f_{ij}(e)$ and then construct the constraint defined by equations 5.3 with the help of equation 5.1 and *LP solver*, where x_{ij} and x_L are the variables.
2. Construct the second constraint defined by the equation 5.2 and then the objective function given in the equation 5.4.
3. Obtain a solution for x_{ij}, by removing the integral constraint. Let this solution be α.
4. Now randomize this α value to get the x value. Here, α_{ij} is using as a probability, to choose the value for x_{ij}, as 1 or 0. This randomization, to be done for all x_{ij} independently, but with the constraint in Equation 5.2.
5. Calculate the new value for $U(e)$ and then, x_L using the Equation 5.3, and then calculate the new cost by the equation 5.4.
6. Repeat the steps 4 and 5, depending on the time allocation for the problem to generate another solution.
7. Choose the best solution.

A sample result of the randomized routing implementation has been attached in section 10.7. The possibility of getting very near optimal solution is increased by repeating the steps 4 & 5 a number of times and selecting the best solution.

5.3 Detailed Routing

In detailed routing, the routing region is decomposed into a collection of rectilinear subregions. There are normally two kinds of such subregions, channels and switchboxes. Channels are routing regions having two parallel rows of fixed terminals, but switchboxes can have fixed terminals on all four sides of the region.

5.3.1 Channel Routing

A channel is bounded by two parallel boundaries and the terminals are arranged in different columns. For a horizontal channel, fixed terminals are on the upper and lower boundaries and floating can be on the left and right ends. The task of a channel router is to route all the nets successfully in the minimum possible areas. Each row in a channel is called as track. If the channel length is fixed, the area minimization becomes channel width minimization, where number of tracks is called as the channel width. One of the fundamental algorithm for channel routing is Left-edge algorithm.

Left-edge Algorithm

This algorithm uses a top-down row-by-row approach [1]. It uses a fundamental graph, called vertical constraint graph to solve the problem. Vertical constrained graph, is one of the fundamental graph associated with a channel. Here the vertices represents the terminals in the top and bottom boundaries of the channel. Every column i such that $t(i)$ and $b(i)$ are not zeros, introduces a directed edge from node $t(i)$ to $b(i)$, in the vertical constraint graph. Here $t(i)$ and $b(i)$ are the terminals located at the column i, on the top and bottom row respectively. A channel with a top and bottom terminal list and its corresponding vertical constraint graph is shown in Figure 5.3.

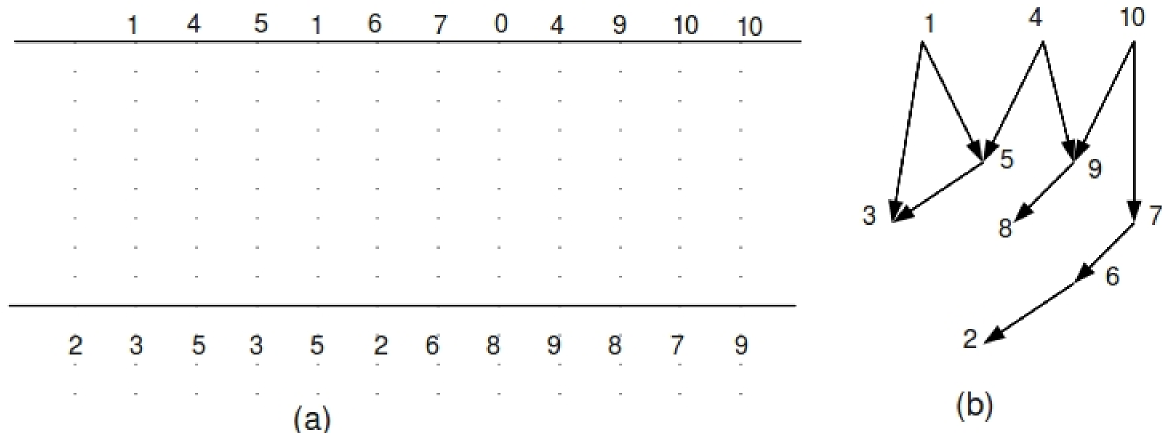

Figure 5.3: (a) A channel with terminal list (b) Corresponding vertical constraint graph

Left edge algorithm, where only one horizontal and one vertical layer are allowed, can be summarized as,

Input: Terminal list in the top and bottom boundary.

Steps:

1. Construct horizontal segment for each net, from the starting and ending columns of the nets.
2. Arrange the nets from left to right according to the starting of horizontal segments. Let the sorted netlist be N.
3. For the given top and bottom terminal list construct vertical constraint graph.
4. calculate nets that has no ancestors from VCG.
5. Choose a net from this no-ancestor list, which has highest priority in the sorted list N, and allot a track. The next no-ancestor one with highest priority can be placed on the same track, if there is no overlapping with the previous net. Repeat this for all the nets in the no-ancestor list.
6. Now update the vertical constraint graph such that the terminals of placed nets, can be removed.
7. Repeat steps 4 and 5, until the tracks are allotted for all the nets.

An example is illustrated in Figure 5.4. In the first iteration, no ancestors exists for the nets $\{1, 4, 10\}$. Since $net - 1$ has highest priority, place this in $track - 1$. Since the segment of next priority one, $net - 4$, is overlapping with the segment of $net - 1$, it cannot be placed. But $net - 10$ can be placed on the same track. Since $net - 1$ and $net - 10$ are placed, remove these terminals from the VCG. After updating the VCG, the nets which has no ancestors are $\{4, 7\}$. The $net - 4$, which has highest priority, can be placed on $track - 2$, but not $net - 7$, since it is overlapping with $net - 4$. This process continues till all the nets are placed. Track-1 is near to the top boundary of the channel. A sample result of the implementation has been attached in section 10.8.

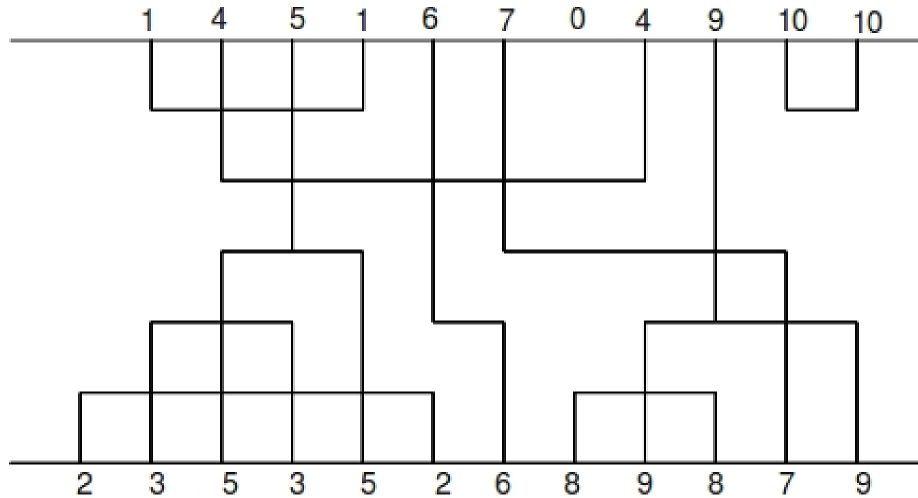

Figure 5.4: Example of result of left edge algorithm

5.4 Detailed Routing Of Standard Cells Using Side Channel

In the standard cell architecture, rectangular cells of same height but variable width, are placed in different rows. The space between two rows is called a *channel* and is used to perform interconnections between cells. At the end of the placement stage, the location of each cell in a row is fixed. But the height of the channel can be changed, by varying the distance between the adjacent cell rows, to accommodate the wires assigned by a global router. If the terminals of the cells to be connected are belonging to different channels, then their interconnection wires passes through *side channel*. Since the width of the side channel can also be changed, a feasible solution is guaranteed.

These interconnections are done in two steps. In the first step, terminals are assigned for the side channel and then, the detailed routing is done for each channel. Figure 5.5 shows an example of the detailed routing in a standard cell layout with side channels. A sample result of the implementation of the algorithm has been attached in section 10.9.

The algorithm can be summarized as follows with netlist as the input and track allotment as the output.

Steps:

1. Construct the top terminal and bottom terminal lists of each channel (except side channel) from the cell list and the netlist. From figure 5.5, top terminal and bottom terminal list of channel-1 are: $\{3, 4, 0, 4, 2, 0, 0\}$ and $\{2, 4, 4, 3, 0, 6, 0\}$ respectively.

2. Construct the list of the terminals in each channel (netlist). In the example, netlist of channel-1 is $\{3, 4, 2, 6\}$.

3. Compare the netlist of each pair of channels to calculate the side terminals of each channel. Thus side terminals of channel-1, in the example are $\{6, 3, 2\}$.

4. Construct the top terminals of the side channel from the side terminals of each channel and

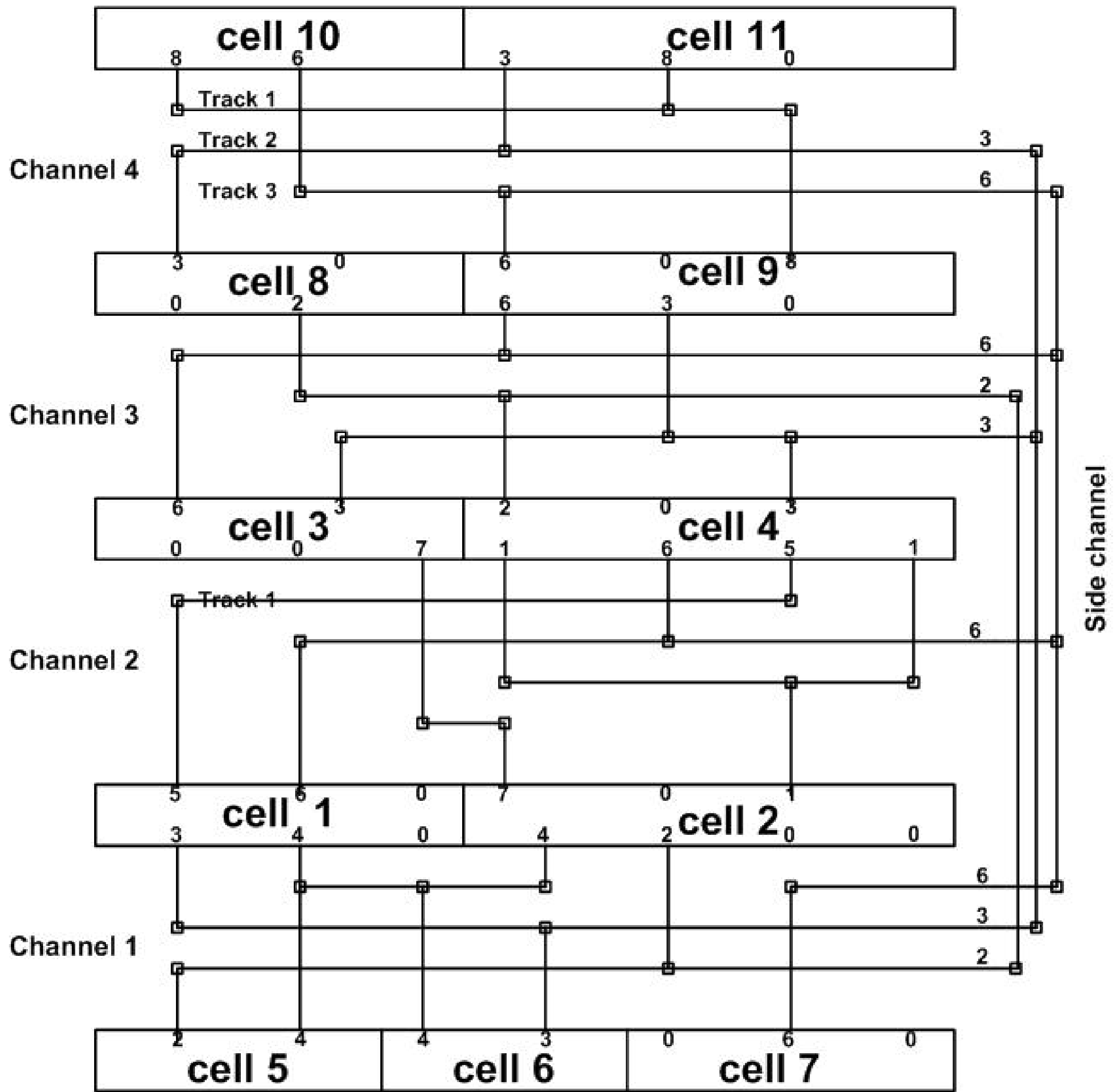

Figure 5.5: Detailed routing of standard cells using side channel

then construct the list of the terminals for the side channel. Here, top terminals of side channel are $\{2, 3, 6, 6, 3, 2, 6, 6, 3\}$ and its netlist is $\{2, 3, 6\}$

5. For each channel perform the following.
   (a) Calculate the ancestor of each net from the top and bottom lists. If circular vertical constraint problem exists, shift the cells in the same row. (If ancestor of net b = a and that of net a = b, then circular vertical constraint problem exists)
   (b) Perform left edge algorithm, which will give the required track allotment.

For the channel-1 in figure 5.5, the $track - 1$ is allotted for net-4 and net-6 , $track - 2$ for net-3 and $track - 3$ for net-2.

Chapter 6

Timing Improvisation

6.1 Introduction

The path between an input and an output with the maximum delay is called as critical path. The delay of such paths needs to be reduced so as to meet the timing requirements, and this can be achieved in number of ways. We have concentrated on one way to handle this problem, in which the buffers can be inserted along the path to strengthen the signal and hence reduce the delay. In this case, the segments at which the buffers can be inserted to be determined.

6.2 Calculation of buffer insertion segments

Calculation of specific segments in the critical path in which the buffers can be inserted, to reduce the delay, is subjected to some conditions. Here, considering the following conditions,

1. The segments, which will be cut, should not be inside dense clusters. This condition can be formulated by calculating the cost associated with the cutting of each segment in the critical path. Cost of cutting of a segment depends upon the number of other segments, which will be affected by its cutting. This cost can be calculated based on *network flow technique.*

2. All consecutive segments, which will be cut, should be within a specified minimum distance, $minD$, and a maximum distance, $maxD$. This can be formulated by introducing some artificial zero cost segments between the pairs of nodes in the critical path, which are $minD$ to $maxD$ apart. Call these artificial segments as *red* lines and original one as *blue* lines.

Find out a shortest path from the starting node to the ending node of the critical path by passing alternatively the *blue* and *red* lines. Since *blue* and *red* lines are coming alternatively, *blue* segments in the shortest path are *red* segments apart. But the length of *red* segments are between $minD$ to $maxD$, thus the buffer insertion segments are within a specified distance. Since the cost of *red* lines are zero, *blue* lines selected for the shortest path will have lowest cost. This ensures that the segments, to be cut, are not inside dense clusters. The shortest path calculation can be done by $Dijkstra's\ algorithm$.

6.2.1 Algorithm to calculate the cost of the segments in the critical path

A graph $G(V, E)$ can be constructed from the given circuit where, V be a set of vertices and E be a set of edges. For each segment in the critical path construct a flow graph G_f from G. The maximum flow value obtained, by performing Ford-Fulkerson or Push-Relabel algorithm on G_f, is the cost of the corresponding segment. This cost calculation algorithm can be summarized as,

Steps:

1. Consider a segment, say S_1, in the critical path. Construct a flow graph G_f for the segment S_1, from the input graph G as follows,

   (a) Add two extra nodes, called *source* and *target*.

   (b) From source, add infinite capacity edges to all the nodes in the critical path, which are on the left side of the segment S_1.

   (c) Similarly, add infinite capacity edges to target, from all the nodes in the critical path, which are on the right of the segment S_1.

   (d) Retain all the edges of original graph G, with capacity one.

2. Perform max-flow calculation algorithm on G_f. Maximum flow value obtained, will give the cost of the segment S_1.

3. Repeat *steps* $1-2$, for each segment in the critical path.

Consider an input graph G, as shown in figure 6.1, with critical path from **node-0** to **node-5**.

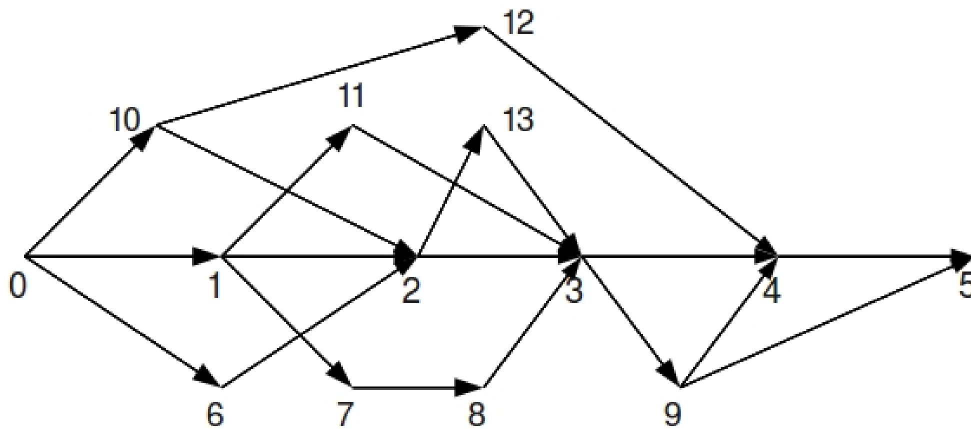

Figure 6.1: Input graph

For a segment from $node-1$ to $node-2$, the flow graph G_f can be constructed as shown in figure 6.2.

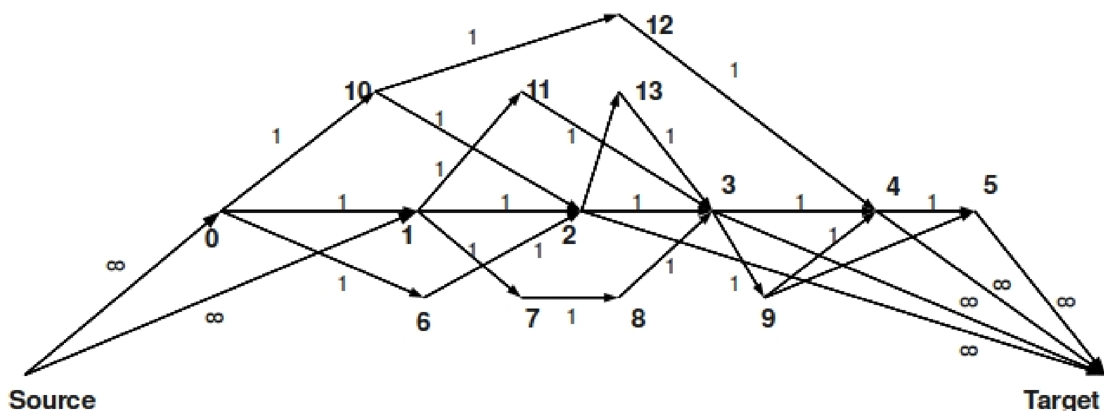

Figure 6.2: Constructed flow graph G_f from G, for the segment $1-2$.

From this flow graph we can see that the maximum flow value is 5 which is the cost of that segment. Similarly G_f can be constructed for each segment and its cost can be calculated. A directed subgraph of G, called G_{sub}, which includes only the critical path is shown in figure 6.3. The calculated cost is shown as the edge weight. This graph G_{sub} is the input for the next algorithm.

Figure 6.3: A subgraph G_{sub}, which includes only the critical path, with calculated cost on the edges

6.2.2 Algorithm to calculate the segments with lowest cost but within specific distance in the critical path

This algorithm is based on *Dijkstra's algorithm* to find a shortest path between a pair of nodes in the directed graph $G(V, E)$, with all edges as nonnegative. Here, the shortest path is calculating on a modified graph with blue and red edges, thus the shortest path will give the segments which are at minimum cost and also within a specific distance. The algorithm can be summarized as follows with the graph G_{sub} as the input .

Steps:

1. Construct a new graph from G_{sub}, called G_{br}, which contains two types of edges called blue and red. Blue edges are the original edges of G_{sub}, with calculated cost as edge weight. Red edges of zero cost are constructing between pair of nodes, which are $minD$ to $maxD$ apart. Graph G_{br} of figure 6.3 is shown in figure 6.4.

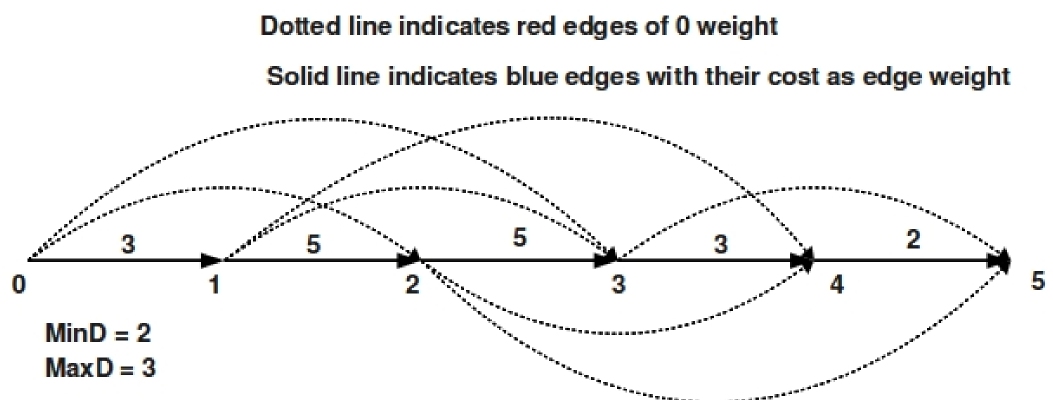

Figure 6.4: Graph G_{br}, which consists of red and blue edges

2. The graph G_{br} to be modified as explained below:
   (a) Insert extra node u' for each node $u \in V$.
   (b) If the edge $(u, v) \in$ *blue edge*, replace (u, v) by (u', v).
   (c) If the edge $(u, v) \in$ *red edge*, replace (u, v) by (u, v').

   Modified graph of G_{br} is shown in figure 6.5.

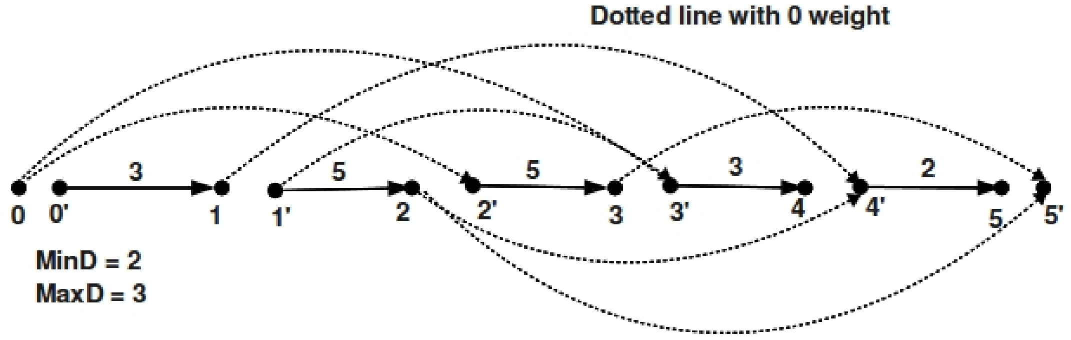

Figure 6.5: Modified Graph of G_{br}

3. Perform *Dijkstra's algorithm* on the modified graph with s and s' as source nodes and t and t' as target nodes. The algorithm can be summarized as follows:
   (a) Initialize the distance and predecessor values of each node, other than source nodes, as $infinite$ and -1 respectively. The distance and predecessor values of source nodes are initialized as 0.
   (b) Insert s and s' in the $Queue, Q$.
   (c) Extract and then delete, the node with minimum distance v, from Q.
   (d) Calculate all the adjoint nodes w, of v.
   (e) For each node w, if the distance of node w is more than the distance of node v + weight of the edge (v, w), then, do the following.
   distance of node w = distance of node v + weight of the edge (v, w).
   predecessor of node w = v.
   Insert w in the $Queue, Q$.
   (f) Repeat steps (c) to (e), till the *Queue* is empty.
4. Calculate the distance of target node t as the smaller of the distances corresponding to t and t', and the corresponding path as the shortest path.
5. Calculate the blue edges in the shortest path, from the *predecessor* list, which are the segments of lowest cost, and thus, the buffers can be inserted in these blue segments.

By performing this algorithm on the modified graph, shown in figure 6.5, calculated nodes in the shortest path, from source 0 to target 5, are $\{0', 1, 4', 5\}$. Edges on the shortest path consists of blue and red edges alternatively. Blue edges in this path are $\{Edge\ from\ 0'\ to\ 1, Edge\ from\ 4'\ to\ 5\}$. Thus the corresponding edges in the original graph, where buffers can be inserted are $Edge\ from\ 0\ to\ 1$ and $Edge\ from\ 4\ to\ 5$. Implementation result has been attached in section 10.10.

Chapter 7

Partitioning

7.1 Partitioning Problem Formulation:

A complex system can be designed efficiently by decomposing into a set of smaller subsystems. The decomposition process is called as partition. The input to the partitioning algorithm is a set of components and a netlist and output is the set of subcircuits. One of the most important objectives in partitioning is the minimization of the number of interconnections between the subsystems or the number of nets cut by the partition.

A graph $G(V, E)$ can be constructed from the given circuit where, V be a set of vertices and E be a set of edges. Each vertex represents the components and edges joining these vertices represents the connections between the components corresponding to these vertices. The partitioning problem is to partition the vertices V into $V_1, V_2, ..., V_k$ where,

$$V_i \cap V_j = \emptyset, \;\; i \neq j$$

$$U_{i=1}^{k} V_i = V$$

The minimization of the cutting edges is a very important objective function for partitioning algorithms since it reduces the delay and the interface between the partitions. Considering a bipartition, weight of the cutting edges to be minimized can be calculated as:

$$\sum_{i \in V, j \in V} W_{ij} * x_i(1 - x_j) \;\; i \neq j$$

where, W_{ij} = weight of the edges from i to j and,

$$\begin{aligned} x_i &= 1 \;\; if \;\; i \in partition - 1 \\ &= 0 \;\; if \;\; i \in partition - 2 \end{aligned}$$

7.2 A New Partitioning Approach:

Circuit partitioning is a fundamental problem in many areas of VLSI physical design thus various approaches are developed. In this method, we are proposing a new partitioning approach, where the number of nodes and edges of the graph to be partitioned, depends only upon the number of flip-flops in the circuit.

A graph $G(V_n, V_f, E)$ is constructed from the given circuit, where V_f and V_n are set of nodes corresponding to flip-flops and to the remaining components respectively. From the constructed graph, the number of common nodes $\in V_n$ between each pair of flip-flops and the amount of direct connection between each flip-flop pairs to be calculated. By using these data, the *connectivity* between

each pair of flip-flops can be determined. The connectivity between a pair of flip-flops is defined to be the sum of x times the number of common nodes and y times the number of direct connections between the flip-flop pair. We can construct a new undirected graph, where the nodes are flip-flops and the weight of the edges represents their connectivity. Since the size of the new graph is very small as compared to the original graph, various partitioning approaches can be applied without much complexity, which will finally give a partition for the original circuit. The algorithm can be summarized as,

Input: Graph of a circuit by specifying the nodes related to flip-flops.

Steps:

1. Modify the flip-flops such that its inputs are now secondary outputs and its outputs are secondary inputs, and thus construct a directed acyclic graph. The idea is illustrated in figure 7.1.
2. Enumerate paths from all the inputs (primary and secondary) to all the outputs by the path enumerating algorithm, which is explained later.
3. Create a table which shows the list of nodes connected to each flip-flop with the help of the enumerated paths.
4. Search the above mentioned table, and construct two matrices. First one shows the number of nodes common to each pair of flip-flops and the second one shows the amount of direct connectivity between the flip-flop pairs.
5. Calculate a cost matrix, based on these matrices. Convert this matrix into a graph, where the nodes are flip-flops and weight of the edges between them is the cost of separating them.
6. Partition the flip-flops, such that they are sufficiently balanced with minimum cutting edge weight, by using any of the partitioning methods. The methods explained here are, simulated annealing, r-balanced flow method and branch and bound method.

A sample result of the implementation of the above algorithm with simulated annealing has been attached in section 10.11.

7.2.1 Path Enumerating Algorithm

This algorithm traverse a directed acyclic graph, which has many input and output nodes, and finds all possible paths from inputs to outputs. The algorithm can be summarized as follows.

Steps:

1. Initialize a stack and do the following for finding the paths.
   (a) Push an input node into the stack.
   (b) Extract and delete the last node v, from the stack and add v to a list of nodes of a path. Calculate its adjoint nodes w, and push them also to the stack.
   (c) If the number of adjoint of v are x, where $x > 1$, then increase the counter called, *number of junctions* by x and push v in the *junction nodes* stack by x times.
   (d) Repeat step $(b) - (c)$ till v is an output node.

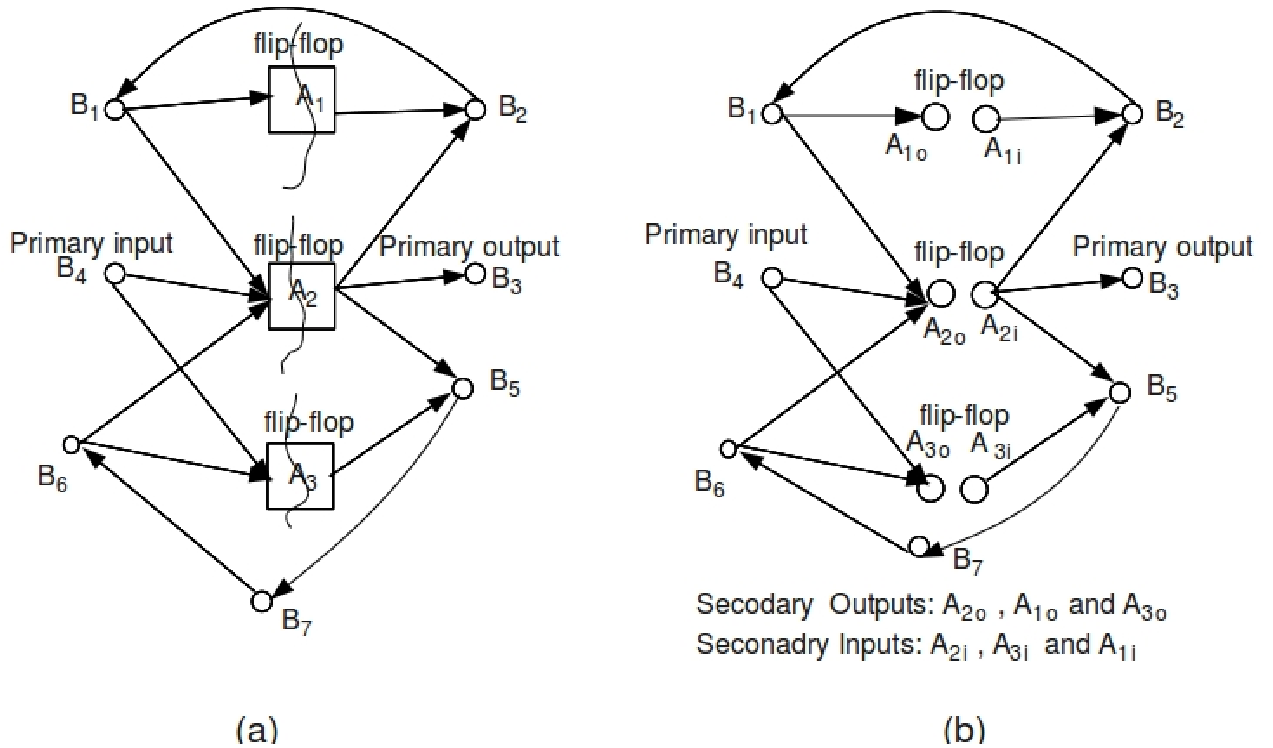

Figure 7.1: (a) Original Graph. (b) Graph after the modification of flip-flop nodes.

(e) If the *number of junctions* > 0, then go back to the previous junction, modify the list of the nodes in the path such that, it contains nodes up to that junction node only, and decrease the *number of junctions* by 1.

(f) Repeat step $(b) - (f)$ till the stack is empty.

(g) Repeat step $(a) - (g)$ for all inputs.

7.2.2 Partitioning by Simulated Annealing

Simulated annealing algorithm, which is based on the annealing process for growing crystals, starts with a random solution. Depending upon the cost calculation and the temperature based probability function, it moves from one solution to another. Trapping in a local minima is avoided by accepting some higher cost solution (bad solution) at high temperature. The probability of accepting a bad solution decreases with cooling of temperature.

Initial value of temperature and the cooling schedule determines the quality of the solution, but time required to generate the solution is proportional to the steps in which the temperature is decreased. Algorithm can be summarized as,

Steps:

1. Start with an initial random solution at high temperature, T.

2. Initialize *count* as zero.

3. Generate new solution from the current one and calculate $\Delta cost$, where,

$$\Delta cost = cost\ of\ new\ solution\ -\ cost\ of\ current\ solution.$$

Replace the current solution with the new one if,

$$\Delta cost < 0, \quad or \quad e^{-\Delta cost/k_B T} > random(0,1).$$

Where, k_B is Boltzmann constant, T is the current temperature, and $random(0,1)$ is a random number between 0 and 1.

4. Increase the count by 1.

5. If the count is less than some specific value repeat 3-4.

6. Update temperature by, $T = T * \alpha$ where, $0 < \alpha < 1$. Here, considered $\alpha = 0.9$.

7. If the temperature > 0, repeat 2-6.

Any bipartition subset $\{V_1, V_2\}$ of given set of nodes V can be the solution. New solution is generated by exchanging two nodes between the bipartition and also by transferring one node from one partition to another, thus searching a large part of the solution space. Finally, the cost is calculated as, $Cost = wt(cutting\ edges\ between\ V_1\ and\ V_2) - a|V_1| - \frac{b}{|V_1|}$, where $|V_1|$ is the number of nodes in the subset V_1. This scheme allows unbalanced partitions as solutions, but the cost will be more for very large and very small set, thus the final result will be sufficiently balanced.

7.2.3 Partitioning by r-balanced Flow Method

According to submodular function property, a good partition, which has minimum number of cutting edges, can be obtained, by minimizing partition associate of the submodular function. By the minimization of partition associate of its zero singleton submodular function(z.s.s.), a non-trivial solution can be obtained. Fusion set (dense set) of this z.s.s. function is contained in one of the blocks of such minimized partition. Finding a subset, which minimizes the zero singleton submodular function over all subsets, is an approach to identify this dense set, and network-flow formulation can be used for this.

Important Definitions:

Submodular function: A function $f : 2^s \rightarrow R$ is submodular iff,

$$f(X) + f(Y) \geq f(X \cup Y) + f(X \cap Y), \;\; \forall X, Y \subseteq V$$

Zero Singleton Submodular function: If a function $f : 2^s \rightarrow R$ is submodular, then its corresponding zero singleton submodular function can be given as,

$$\hat{f}(U) = f(U) - \sum_{u \in U} f(u)$$

with following properties:

- $\hat{f}(U)$ is submodular.
- $\hat{f}(e) = 0, \;\; \forall e \in S$

Fusion Set: If $f(.)$ is a z.s.s. on subsets of S, then a set $T \subseteq S$ is called as a fusion set of f iff:

- $f(T) < 0$,
- $f(T) \leq f(R), \;\; \forall R \subseteq T$ and

- All subsets of T, on which $f(.)$ achieves a negative value, have a common element.

By using the following theorem we can find the fusion set from z.s.s..

Theorem: If $f(.)$ is a z.s.s. on subsets of S, and N be a fusion set of $f(.)$, then there exists a partition of S such that partition associate of $f(.)$ reaches a minimum on it and N is contained in one of the blocks of the partition.

Consider a function $|\Gamma|(U)$, where $\Gamma(U)$ is the set of all edges, which have at least one end-point inside U is submodular and $|\Gamma|(\emptyset) = 0$.

Its bipartition associate $\Gamma(U)'$ is defined as,

$$\Gamma(U)' = |\Gamma|(V_1) + |\Gamma|(V_2).$$

It is clear that,

$$\Gamma(U)' = \textit{ set of all edges of graph } G \;+\; \textit{two times cutting edges}$$

and thus minimizing $\Gamma(U)'$ implies the minimization of the cutting edges.

But since $|\Gamma|(U)$ is a submodular function, by definition,

$$|\Gamma|(V_1) + |\Gamma|(V_2) \geq \Gamma(V_1 \cup V_2) + \Gamma(V_1 \cap V_2).$$

Since $V_1 \cap V_2 = \emptyset$ for the partition, $\Gamma(V_1 \cap V_2) = 0$. Thus,

$$|\Gamma|(V_1) + |\Gamma|(V_2) \geq \Gamma(V_1 \cup V_2)$$

where $\Gamma(V_1 \cup V_2)$ is full set and the minimization of $\Gamma(U)'$ is trivial.

A non-trivial solution can be obtained by using a modular function $g(U)$ with the submodular function, and thus getting the zero singleton submodular function. Since the partition associate of the modular function with $g(\emptyset) = 0$ remains constant irrespective of the partition, minimizing the partition associate of the zero singleton submodular function is the new problem to be solved. By identifying a dense set, which minimizes the z.s.s., this required minimizing partition will be obtained.

Consider a submodular function $-|\tilde{E}|(U) - k$, where $\tilde{E}(U)$ is the set of all edges, which have both the end points inside U. k is used in the function to obtain a non-trivial solution.

Its bipartition associate is,

$$-|\tilde{E}|(V_1) - |\tilde{E}|(V_2) - 2k = -(\textit{ set of all edges of graph } G \;-\textit{cutting edges }) - 2k.$$

Thus the partition minimizing this partition associate reduces the net-cut. It can be obtained from the dense set, which minimizes z.s.s. of $-|\tilde{E}|(U) - k$.

The z.s.s. is defined as,

$$-|\tilde{E}|(U) - k - \sum_{u \in U}(-k) = k|U| - |\tilde{E}|(U) - k.$$

Minimizing this over all possible subset is equivalent of minimizing,

$$\min_{U \in V}(k|U| - |\tilde{E}|(U)),$$

which can be obtained by network flow formulation.

Network Flow Formulation:

A given graph $G(V, E)$ can be converted to a bipartite graph $G_b(V_n, V_e, E_b)$, where,

1. $V_n \equiv$ Set of vertices, which has one vertex corresponding to each vertex of original graph G.
2. $V_e \equiv$ Set of vertices, which has one vertex corresponding to each edge of original graph G.
3. $E_b \equiv$ Set of directed edges, which connect a vertex $x \in V_n$ to a vertex $y \in V_e$, iff in the graph G, the edge corresponding to the node y, is incident on the vertex corresponding to the vertex x.

Convert this bipartite graph into a flow graph G_f as follows:

1. Add a source vertex s and construct edges from it to all nodes from the subset V_n with a capacity k.
2. Add a target vertex t and construct edges from all the nodes from subset V_e to it, with capacity equal to the weight of the corresponding edges in the original graph G.
3. Capacity of the original bipartite edges E_b, in the flow graph G_f is infinity.

An input graph G and its initial flow graph G_f is shown in figure 7.2 When max-flow is reached, all the forward edges from source-side to target-side are saturated and reverse edges carry *zero* flow. A typical flow graph when max-flow is reached, is shown in figure 7.3. Here,no edges exists from

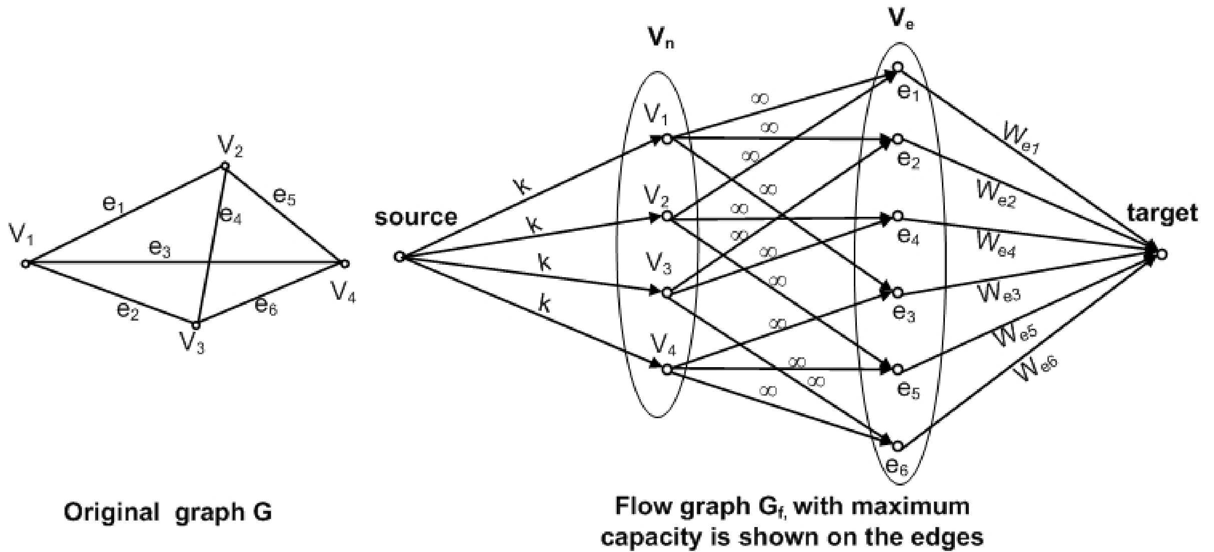

Figure 7.2: Graph G and its corresponding flow graph G_f.

U to $V_e - P$ and the edges from $V_n - U$ to P should carry *zero* flow, since max flow is reached. But since the edges from P to *target* are saturated, there exists no nodes in P which is not connected to any nodes in U. Thus,

$$P \cong |\Gamma|(U)$$

and by considering $W_e = 1$,

$$Max\ flow = |\Gamma|(U) + k|V_n - U| = |V|_e - |\tilde{E}|(V_n - U) + k|V_n - U|$$

and

$$Capacity\ of\ min\ cut = |V|_e - |\tilde{E}|(V_n - U) + k|V_n - U|$$

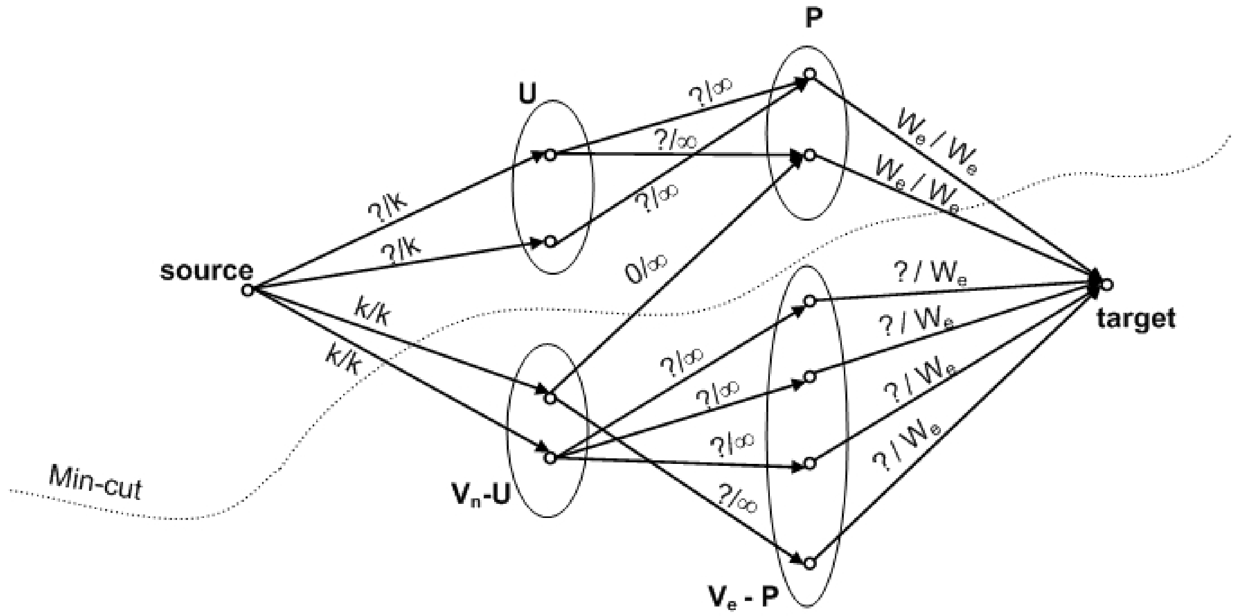

Figure 7.3: A typical flow graph when max-flow is reached

where, U belongs to the source side subset of V_n. Since the above equation is the capacity of min cut, for any $U \subseteq V_n$, max-flow minimizes

$$\min_{U \subseteq V}(|V|_e - |\tilde{E}|(U) + k|U|.$$

Since $|V|_e$ is constant, max flow minimizes

$$\min_{U \subseteq V}(k|U| - |\tilde{E}|(U).$$

which gives the required dense set U.

r-balanced flow based bipartition algorithm:

The algorithm can be summarized as,

1. Convert the given graph $G(V, E)$ into a bipartite graph $G_b(V_n, V_e, E_b)$ and into a flow graph G_f as explained before.

2. Perform the Ford-Fulkerson or Push-Relabel algorithm on G_f, and calculate the minimum cut, which gives the clustering {X′, $\tilde{\mathrm{X}}'$ }, where X′ ∈ source and $\tilde{\mathrm{X}}'$ ∈ target side.

3. Calculate the nodes of original graph, X ∈ X′. Check for the condition, (1 - ϵ)rW ≤ w(X) ≤ (1 + ϵ)rW, where, w(X) is weight of nodes in the set X , W is the total node weight of original graph, ϵ is the deviation factor of weight and r value can be 0.5. Then this bipartition is sufficiently balanced, and thus the returns the result {X, $\tilde{\mathrm{X}}$}. Else go to the next step.

4. If w(X) < (1 - ϵ)rW, then collapse all nodes in X′ to source s. Also collapse an additional node, v to s, where v ∈ $\tilde{\mathrm{X}}'$ and also a part of the minimum cut. Construct the new flow graph from the previous one by retaining the flow value and go to $step - 3$.

5. If w(X) > (1 + ϵ)rW, then collapse all nodes in $\tilde{\mathrm{X}}'$ to target t. Also collapse an additional node, v to t, where v ∈ X′ and also a part of the minimum cut. Construct the new flow graph from the previous one by retaining the flow value and go to $step - 3$.

By collapsing v to s, by retaining the previous source and target nodes, the algorithm is able to explore a different netcut with larger X in the next iteration. Similarly by collapsing v to target, by retaining the previous source and target nodes, a smaller X will be getting in the next iteration.

Selection of the node v, from the the set of nodes incident on the min-cut, should be done carefully, if the remaining circuit is small. If the remaining nodes in the circuit is smaller than a threshold value, ideally, we should try all the nodes incident on the min-cut, and pick the node whose collapsing induces a min-net-cut with the smallest size.

After the clustering, the previous flow values are retaining in $step-5$ and $step-6$ of the algorithm. Thus in each iteration, the additional flow to saturate the edges between the source cluster and the target cluster only have to be calculated. Then the total complexity is almost of the same order as that of a single maxflow-mincut computation.

7.2.4 Partitioning by Branch and Bound method

Branch and Bound algorithm systematically searches the complete space of solutions for a given problem for the best solution. First step of this algorithm is to define a solution space and then organize it so that it can be searched easily. In this algorithm, searching is done in breadth first manner, beginning from the root node, which is both *live* node and *E-node* (*Expanding node*). From this E-node, all the nodes that can be reached using a single move are generated, but the nodes that cannot lead to a feasible solution are discarded. The remaining nodes are added to the list of live nodes, in the form of a queue. The next E-node can be selected from the queue, in the $FIFO$ order and the expansion process is continued until the queue becomes empty.

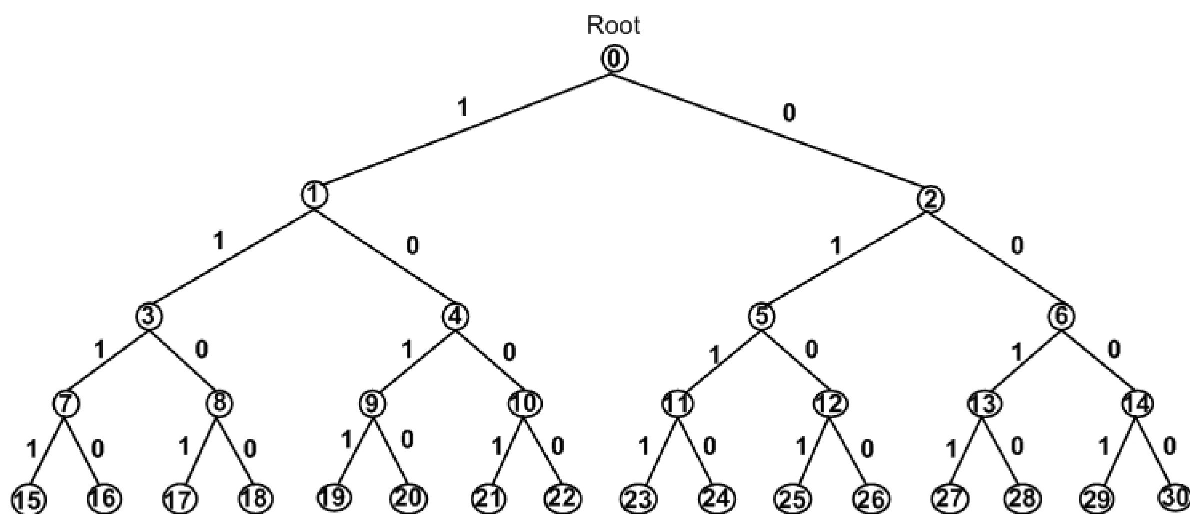

Figure 7.4: A solution space structure for partitioning a set of four vertices

Partitioning Application

For a partitioning problem, the solution space organization can be considered like a binary tree as shown in Figure 7.4. The nodes are arranged in different levels with root-node at level-1 and the nodes from $15-30$, which are called as leaf nodes are at the last level. All paths from the root to a leaf node define a solution space. The label on an edge from a node-level i to $i+1$ indicates the corresponding vertex position in the bipartition. A solution [1,1,0,0], which indicates first two vertices in partition-1 and remaining in partition-2, is defined by a path from the root to leaf node 18. The algorithm can be summarized as,

1. Initialize the node-level as 1 and begin the search from the root node 0, by inserting it in the FIFO queue, and then insert -1 to indicate the end of current node-level.

2. Generate the left child node of the E-node, which has extracted from the FIFO queue, and if it is feasible, consider as a live node and insert in the queue, otherwise discard it. It can be considered as feasible, if the number of *ones* up to this node from the root is less than or equal to the maximum possible size of the first partition.

3. Similarly generate the right child, and if it is feasible, consider as a live node and insert in the queue. But here the feasibility checking is by counting the number of *zeros* up to this node from the root, and comparing it with the maximum size of the second partition.

4. Extract the first element from the queue, and if it is not equal to -1, repeat steps 2-4.

5. If it is -1, which indicates all the live nodes from the current node-level is extracted, add an end-of-level marker, -1, to the queue, increase the node-level by 1, and extract the first element from the queue. Repeat steps 2-5 if the node-level is less than the number of elements in the original set to be bi-partitioned.

6. If the node-level is equal to the number of elements, child nodes of the E-node will be the *leaf* nodes, thus no need to be inserted in the queue. But generate the left and right leaf nodes, and get the solution by tracing the path from the leaf to the root node with the help of *predecessor* list.

7. Accept the solution if the size of the partition is feasible, calculate the cutting edges and then the cost corresponding to this solution.

8. If it is less than the cost of the previous solution update this as the best solution.

9. Repeat steps 6-8 until the extracted element from the queue is -1.

Cost can be calculated similar to the simulated annealing method, which will give sufficiently balanced solution. The time and space requirement for this algorithm depends upon the number of leaf nodes (size of solution space organization) and thus is $O(2^n)$.

A sample result of the partitioning implementation by simulated annealing, r-balanced flow based method and branch and bound method, with the same input, has been attached in Section 10.12.

Chapter 8

Conclusion

We have discussed many of the important algorithms in VLSI physical design. In the floorplanning section, an algorithm by Otten and Stockmeyer for the minimum area implementation of the sliceable floorplan is an important one.

Placement approaches such as simulated annealing, genetic algorithm, regular placement assignment problem and also algorithm based on linear resistive network have been discussed. Simulated annealing operates only on one solution at a time, while genetic algorithm maintains a large population of solutions which are optimized simultaneously. Thus genetic algorithm can be efficiently parallelized. Both the approaches employ techniques to avoid entrapment at local minima. In regular placement assignment problem, all the targets are predetermined and uses the linear programming approach. In the placement algorithm based on linear resistive network an electrical analogy is considered where module position is analogous to the node voltage. A widely used algorithm for timing-driven placement, called Zero slack algorithm, is also discussed. In this algorithm maximum allowable delay of each net is calculated based on which timing driven placement could be done.

In the routing section, maze running and steiner tree algorithms are presented, which serve as the foundation for many global and detailed routing algorithms. A randomized routing algorithm based on linear integer programming, is also discussed. We have proposed and implemented an algorithm for the detailed routing of standard cell with side channel. Here left edge algorithm is used for the routing of each channel. Since the width of the side channel can be changed in this approach, a feasible solution is guaranteed.

A timing improvisation method is also proposed and implemented, where the delay of the critical path can be reduced by inserting buffers in some specific critical path segments. In this approach, the segments which are within specific distance apart, but not inside dense clusters, are considered. An approach based on Dijkstra 's algorithm and network flow technique are employed for determining these segments.

We have also proposed a new approach for circuit partitioning. In this approach, a new graph is constructed from the original, where the nodes represent the flip-flops and the edges represent their respective connectivity. Since the size of the new graph is very small, various partitioning approaches can be applied with less complexity. In this, simulated annealing approach, r-balanced flow based method and branch and bound method, have been discussed.

A lot of ongoing work of fellow students in the group has involved netlist extraction from RTL description of the circuit and developing algorithm for circuit synthesis using technology mapping.

Integrating all the above work would pave the way for the development of new efficient algorithms in different areas of VLSI physical design.

Chapter 9

Future Work

The algorithms worked out and implemented in the project will be benchmarked using larger VLSI circuits. Many of these algorithms can be implemented efficiently for parallel processing. By providing a foundation for VLSI CAD in the group, the project has opened leads for a lot of potential work. More on FF-aware partitioning has been worked out and now needs to be implemented. In the long run, the group is also aiming at designing its own analog of FPGA boards.

Chapter 10

Results

10.1 Optimal area implementation for a sliceable floor plan

Implementation file saved as *final_sliceable_floorplan.c*
INPUT
How many modules? : 5
How many edges(which represents connectivity)? : 7
Connectivity graph is inputting from the file *sdata1*

OUTPUT
Sorted Edges of the connectivity Graph
v w wt is 1 3 50
v w wt is 2 4 50
v w wt is 2 3 35
v w wt is 1 2 15
v w wt is 3 4 15
v w wt is 4 5 13
v w wt is 3 5 10
We are clustering the nodes 1 and 3.
Enter two possible width options in ascending order for the module 1 : 4 9
Enter two possible height options in descending order for the module 1 : 9 3
Enter two possible width options in ascending order for the module 3: 6 8
Enter two possible height options in descending order for the module 3: 8 2
Performing **vertical node sizing** while merging the nodes 1 and 3
All possible width options calculated after performing VNS for the merged node : 10 15 17
All possible height options calculated after performing VNS for the merged node: 9 8 3

We are clustering the nodes 2 and 4.
Enter two possible width options in ascending order for the module 2 : 4 8
Enter two possible height options in descending order for the module 2 : 7 3
Enter two possible width options in ascending order for the module 4: 4 7
Enter two possible height options in descending order for the module 4: 9 2
Performing **vertical node sizing** while merging the nodes 2 and 4
All possible width options calculated after performing VNS for the merged node : 8 11 15
All possible height options calculated after performing VNS for the merged node: 9 7 3

We are clustering the nodes 13 and 24.
Performing **horizontal node sizing** while clustering the nodes 13 and 24

All possible width options calculated after performing HNS for merged node : 10 15 17
All possible height options calculated after performing HNS for merged node: 18 17 12

We are clustering the nodes 5 and 1324.
Enter two possible width options in ascending order for the module 5 : 5 6
Enter two possible height options in descending order for the module 5 : 7 2
Performing **vertical node sizing** while merging the nodes 5 and 1324
All possible width options calculated after performing VNS for the merged node : 15 20 22
All possible height options calculated after performing VNS for the merged node: 18 17 12

Minimum required area is 270.

10.2 Placement of Standard cell by Simulated Annealing

Implementation file saved as *placement_one_dim_sa.sce*
Start up scilab and execute the file at the scilab_prompt > *exec placement_one_dim_sa.sce*
Run the function at the scilab_prompt > *place_one_dim*
Input
Enter the total number of modules : 7
How many rows are there to arrange the modules ?:3
Enter the number of modules in Row 1 : 3
Enter the number of modules in Row 2 : 2
Enter the number of modules in Row 3 : 2
Enter the width of modules : 12 11 13 14 15 9 10
Enter the height of the modules : 21
Enter the number represents connectivity between 1 and others : 2 3 2 4 5 0
Enter the number represents connectivity between 2 and others: 9 4 2 1 4
Enter the number represents connectivity between 3 and others: 5 0 2 1
Enter the number represents connectivity between 4 and others: 3 4 2
Enter the number represents connectivity between 5 and others: 3 1
Enter the number represents connectivity between 6 and 7 : 2
Output
optim_sa: Temperature step 1 / 20 - T = 273.330655, E(f(T)) = 1934.700000 var(f(T)) = 98.348643
f_best = 1786.000000
optim_sa: Temperature step 2 / 20 - T = 245.997590, E(f(T)) = 1835.100000 var(f(T)) = 78.967574
f_best = 1702.000000
optim_sa: Temperature step 3 / 20 - T = 221.397831, E(f(T)) = 1794.600000 var(f(T)) = 35.132132
f_best = 1702.000000
optim_sa: Temperature step 4 / 20 - T = 199.258048, E(f(T)) = 1866.200000 var(f(T)) = 140.748633
f_best = 1702.000000
optim_sa: Temperature step 5 / 20 - T = 179.332243, E(f(T)) = 1991.500000 var(f(T)) = 158.225753
f_best = 1702.000000
optim_sa: Temperature step 6 / 20 - T = 161.399019, E(f(T)) = 1877.800000 var(f(T)) = 124.811680
f_best = 1702.000000
optim_sa: Temperature step 7 / 20 - T = 145.259117, E(f(T)) = 1798.800000 var(f(T)) = 106.360811
f_best = 1624.000000
optim_sa: Temperature step 8 / 20 - T = 130.733205, E(f(T)) = 1939.100000 var(f(T)) = 96.901382
f_best = 1624.000000
optim_sa: Temperature step 20 / 20 - T = 36.922918, E(f(T)) = 1817.900000 var(f(T)) = 41.913800
f_best = 1624.000000

Best module location :
module 1 at position : 7
module 2 at position : 2
module 3 at position : 3
module 4 at position : 4
module 5 at position : 5
module 6 at position : 6
module 7 at position : 1

Minimum cost = 1624.000000

10.3 Placement of Macro cell by Genetic Algorithm

Implementation file is saved as *gentic_algo_macro_cell_placement.c*
Input is reading from the file : *gentic_data5*

Printing the input data read from file
no. of module is 12
module no is: 1 2 3 4 5 6 7 8 9 10 11 12
widths of modules in order : 20 40 80 50 30 20 70 30 20 70 80 75
height of modules in order: 30 40 40 40 70 50 90 70 50 90 30 20
Connectivity is
:34 :37 :28 :16 :44 :36 :37 :43 :50 :22 :13 :28 :41 :10 :14 :27 :41 :27 :23 :37 :12 :19 :18 :30 :33 :31 :13 :24 :18 :36 :30 :3 :23 :9 :20 :18 :44 :7 :12 :43 :30 :24 :22 :20 :35 :38 :49 :25 :16 :21 :14 :27 :42 :31 :7 :24 :13 :21 :47 :32 :6 :26 :35 :28 :37 :6

Output data
Best position. of modules is
8 5 6 7 9 4 1 2 3 11 10 12
Corresponding orientation of modules is (Orientation=0 means vertical; Orientation=1 means horizontal;
1 1 1 0 1 1 1 0 0 0 10
x-coordinates of the best fit solution in module order is
70 90 0 20 30 60 80 0 0 70 40 140
y-coordinates of the best fit solution in module order is
70 70 120 70 0 0 0 0 70 120 120 120

10.4 Placement by Regular Placement Assignment Problem

Implementation file saved as *gen_regularplacemnet_assignment_f_read.c*
How many modules?:(15) :15
How many fixed modules?(2) :2
How many nets? :(20) 20
Entering each net's starting module, ending module & connectivity (wt) from input file : *place_data1*
Considered starting and ending x values of chip as 0 and 100
Considered starting and ending y values of chip as 0 and 60
Module numbers from 1 to 15 and 2 modules are fixed
Enter the fixed module's module no, fixed X-value (0 to 100) and fixed Y- value(0 to 60) :

module no (0 - 15): 3
X-value (0-100): 20
Y- value (0-60): 30
module no (0 - 15): 5
X-value (0-100): 50
Y- value (0-60): 20
Entering each targets x and y values from the input file
Initial solution with the help of LP solver

Calculation of x-coordinates
/* Objective function */
min: -10 C1 +10 C2 -5 C3 +5 C4 -20 C5 +20 C6 -10 C7 +10 C8 -5 C9 +5 C10 -10 C11 +10 C12 -2 C13 +2 C14 -5 C15 +5 C16 -4 C17 +4 C18 -10 C19 +10 C20 -19 C21 +19 C22 -12 C23 +12 C24 -10 C25 +10 C26 -13 C27 +13 C28 -16 C29 +16 C30 -15 C31 +15 C32 -20 C33 +20 C34 -25 C35 +25 C36 -26 C37 +26 C38 -15 C39 +15 C40;
/* Some of the Constraints */
R2: +C1 $\geq$ 0; R3: +C2 $\leq$ 100;
-C1 +C41 $\geq$ 0; -C1 +C42 $\geq$ 0;
-C2 +C42 $\leq$ 0; -C4 +C41 $\leq$ 0;
R8: +C3 $\geq$ 0; R9: +C4 $\leq$ 100;
-C3 +C41 $\geq$ 0; -C3 +C43 $\geq$ 0;
-C4 +C43 $\leq$ 0; -C6 +C42 $\leq$ 0;
R14: +C5 $\geq$ 0; R15: +C6 $\leq$ 100;
-C5 +C42 $\geq$ 0; -C5 +C43 $\geq$ 0;
-C6 +C43 $\leq$ 0; -C8 +C41 $\leq$ 0;
R116: +C39 $\geq$ 0; R117: +C40 $\leq$ 100;
-C39 +C47 $\geq$ 0; R121: +C43 =20;
R122: +C45 = 50;
Objective value: 780.000000
Module-1's X value is 20.000000 Module-2's X value is 20.000000
Module-3's X value is 20.000000 Module-4's X value is 20.000000
Module-5's X value is 50.000000 Module-6's X value is 50.000000
Module-7's X value is 50.000000 Module-8's X value is 20.000000
Module-9's X value is 20.000000 Module-10's X value is 20.000000
Module-11's X value is 50.000000 Module-12's X value is 50.000000
Module-13's X value is 50.000000 Module-14's X value is 50.000000
Module-15's X value is 50.000000

Similarly Objective function and Constraints constructed for y-coordinates and solved to get
Module-1's y value is 30.000000 Module-2's y value is 30.000000
Module-3's y value is 30.000000 Module-4's y value is 30.000000
Module-5's y value is 20.000000 Module-6's y value is 20.000000
Module-7's y value is 20.000000 Module-8's y value is 30.000000
Module-9's y value is 30.000000 Module-10's y value is 30.000000
Module-11's y value is 20.000000 Module-12's y value is 20.000000
Module-13's y value is 20.000000 Module-14's y value is 20.000000
Module-15's y value is 20.000000

Calculated the distance of each target from each module.
Assignment Done such that the cost is minimum.

Module 1 is assigned to target 1
Module 2 is assigned to target 2
Module 3 is assigned to target 3
Module 4 is assigned to target 4
Module 8 is assigned to target 5
Module 9 is assigned to target 6
Module 5 is assigned to target 7
Module 6 is assigned to target 8
Module 7 is assigned to target 9
Module 11 is assigned to target 10
Module 12 is assigned to target 11
Module 13 is assigned to target 12
Module 14 is assigned to target 13
Module 15 is assigned to target 14
Module 10 is assigned to target 15

Total optimal cost is 367.102753

10.5 Zero Slack Algorithm

Implementation file is saved as *gen_zero_slack.c*
Input is reading from the file, *zdata3*
Number of input nodes = 4 , Number of output nodes = 2 (These values taken from file)

list of input nodes (from file) are : 0 1 2 3
Enter the arrival time at the input node 0 : 10
Enter the arrival time at the input node 1 : 20
Enter the arrival time at the input node 2 : 0
Enter the arrival time at the input node 3 : 0

list of output nodes (from file) are : 9 10
Enter the required time at the output node 9 : 100
Enter the required time at the output node 10 : 90
Given Edges of GRAPH with wt as net delay
v w wt is 0 4 0.000000
v w wt is 1 4 0.000000
v w wt is 2 6 0.000000
v w wt is 2 5 0.000000
v w wt is 3 6 0.000000
v w wt is 4 5 0.000000
v w wt is 5 7 0.000000
v w wt is 6 8 0.000000
v w wt is 6 7 0.000000
v w wt is 7 9 0.000000
v w wt is 8 10 0.000000
Arrival time required time and slack time of each node
node 0 : 10.000000 100.000000 90.000000
node 1 : 20.000000 100.000000 80.000000
node 2 : 0.000000 90.000000 90.000000
node 3 : 0.000000 90.000000 90.000000
node 4 : 20.000000 100.000000 80.000000
node 5 : 20.000000 100.000000 80.000000
node 6 : 0.000000 90.000000 90.000000
node 7 : 20.000000 100.000000 80.000000
node 8 : 0.000000 90.000000 90.000000
node 9 : 20.000000 100.000000 80.000000
node 10 : 0.000000 90.000000 90.000000

total slack is 940.000000

New graph with updated net delay as wt
v w wt is 0 4 0.000000
v w wt is 1 4 20.000000
v w wt is 2 6 0.000000
v w wt is 2 5 0.000000
v w wt is 3 6 0.000000
v w wt is 4 5 20.000000
v w wt is 5 7 20.000000
v w wt is 6 8 0.000000
v w wt is 6 7 0.000000
v w wt is 7 9 20.000000
v w wt is 8 10 0.000000
Arrival time required time and slack time of each node
node 0 : 10.000000 40.000000 30.000000
node 1 : 20.000000 20.000000 0.000000
node 2 : 0.000000 60.000000 60.000000
node 3 : 0.000000 80.000000 80.000000
node 4 : 40.000000 40.000000 0.000000
node 5 : 60.000000 60.000000 0.000000
node 6 : 0.000000 80.000000 80.000000
node 7 : 80.000000 80.000000 0.000000
node 8 : 0.000000 90.000000 90.000000
node 9 : 100.000000 100.000000 0.000000
node 10 : 0.000000 90.000000 90.000000
total slack is 430.000000

This process repeated many times and finally,

New graph with updated net delay as wt
v w wt is 0 4 30.000000
v w wt is 1 4 20.000000
v w wt is 2 6 0.000000
v w wt is 2 5 60.000000
v w wt is 3 6 40.000000
v w wt is 4 5 20.000000
v w wt is 5 7 20.000000
v w wt is 6 8 0.000000
v w wt is 6 7 40.000000
v w wt is 7 9 20.000000
v w wt is 8 10 50.000000
Arrival time required time and slack time of each node
node 0 : 10.000000 10.000000 0.000000
node 1 : 20.000000 20.000000 0.000000
node 2 : 0.000000 0.000000 0.000000
node 3 : 0.000000 0.000000 0.000000
node 4 : 40.000000 40.000000 0.000000
node 5 : 60.000000 60.000000 0.000000
node 6 : 40.000000 40.000000 0.000000

node 7 : 80.000000 80.000000 0.000000
node 8 : 40.000000 40.000000 0.000000
node 9 : 100.000000 100.000000 0.000000
node 10 : 90.000000 90.000000 0.000000
total slack is 0.000000

New graph with updated net delay as wt
v w wt is 0 4 30.000000
v w wt is 1 4 20.000000
v w wt is 2 6 0.000000
v w wt is 2 5 60.000000
v w wt is 3 6 40.000000
v w wt is 4 5 20.000000
v w wt is 5 7 20.000000
v w wt is 6 8 0.000000
v w wt is 6 7 40.000000
v w wt is 7 9 20.000000
v w wt is 8 10 50.000000

10.6 Steiner tree

Implementation file is saved as *final_steiner.c*
Input is reading from the file, *data7*
Total number of vertices are 10, (0-9)
Enter Total no. of Terminal vertices in steiner tree (Enter a number less than 10) : 4
Enter the 4 terminal vertices, one after another (Enter all nodes only between 0 to 10): 0 5 6 7
Edges of Original Graph G
edge from 0 to 3 with wt = 2
edge from 0 to 2 with wt = 1
edge from 0 to 1 with wt = 2
edge from 1 to 4 with wt = 2
edge from 1 to 5 with wt = 9
edge from 1 to 3 with wt = 8
edge from 2 to 9 with wt = 5
edge from 2 to 7 with wt = 3
edge from 2 to 5 with wt = 4
edge from 3 to 6 with wt = 8
edge from 3 to 9 with wt = 5
edge from 4 to 5 with wt = 8
edge from 5 to 7 with wt = 8
edge from 6 to 9 with wt = 8

RESULT
Edges of Graph G1, where the nodes are the terminal vertices and edge wt between each pair of nodes is the shortest distance
edge from 0 to 7 with wt = 4
edge from 0 to 6 with wt = 10
edge from 0 to 5 with wt = 5
edge from 5 to 7 with wt = 7
edge from 5 to 6 with wt = 15

edge from 6 to 7 with wt = 14
Mapping of G1 with G (list of vertices of G for each edges of G1)
Edge 0-5 of G1 : Vertices of G are : 0 2 5
Edge 0-6 of G1 : Vertices of G are : 0 3 6
Edge 0-7 of G1 : Vertices of G are : 0 2 7
Edge 5-6 of G1 : Vertices of G are : 5 2 0 3 6
Edge 5-7 of G1 : Vertices of G are : 5 2 7
Edge 6-7 of G1 : Vertices of G are : 6 3 0 2 7
Spanning tree of G1
edge from 0 to 7 with wt = 4
edge from 0 to 5 with wt = 5
edge from 0 to 6 with wt = 10
Mapping of spanning tree of G1 with G (list of vertices for each edge)
Edge 0-5 : Vertices of G are : 0 2 5
Edge 0-6 : Vertices of G are : 0 3 6
Edge 0-7 : Vertices of G are : 0 2 7
Steiner tree, where terminal vertices are 0 5 6 7 ,
edge from 0 to 2 with wt = 1
edge from 2 to 5 with wt = 4
edge from 0 to 3 with wt = 2
edge from 3 to 6 with wt = 8
edge from 2 to 7 with wt = 3

10.7 Randomized Routing

Implementation file is saved as *final_random_randomized_routing.c*
How many nets ?: 3
Enter the cost of net[1] : 5 Enter the cost of net[2] : 8 Enter the cost of net[3] : 7
Enter the no. of trees for net[1] : 3
Enter the no. of trees for net[2] : 2
Enter the no. of trees for net[3] : 4
Enter the lamda value: which is the cost for exceeding load capacity of channel (enter a value around 25 times that of the cost of net): 200
Random value had taken for the capacity and length of each edges and edges of each trees
Also inputting a random value between 1 to ne for the edges in each tree
/* Objective function */
min: +200 C1 +320 C2 +335 C3 +448 C5 +440 C6 +182 C9 +301 C10 +266 C11 +329 C12 +200 C13;
/* Constraints */
+1.25 C1 +1.25 C2 -C13 ≤ 0; +0.625 C1 +0.625 C2 +0.625 C3 -C13 ≤ 0;
+1.25 C3 +2 C5 +1.75 C12 -C13 ≤ 0; +0.714285746217 C2 +1.14285719395 C5 +1.14285719395 C6 +1.0000000447 C10 +1.0000000447 C12 -C13 ≤ 0;
+0.700000010431 C9 +0.700000010431 C10 -C13 ≤ 0;
+1.66666671634 C1 +1.66666671634 C3 +2.33333340287 C11 -C13 ≤ 0; +8 C5 -C13 ≤ 0;
+2 C5 +2 C6 +1.75 C12 -C13 ≤ 0; +8 C6 +7 C9 +7 C10 -C13 ≤ 0;
+1.66666671634 C2 +1.66666671634 C3 +2.66666674614 C6 +2.33333340287 C9 +2.33333340287 C10 -C13 ≤ 0; +2.5 C1 +3.5 C11 -C13 ≤ 0;
+0.625 C3 +0.875 C11 -C13 ≤ 0; +1.66666671634 C2 +2.66666674614 C6 +2.33333340287 C11 +2.33333340287 C12 -C13 ≤ 0;
+2.33333340287 C10 -C13 ≤ 0; +C1 +C2 +C3 = 1;

+C5 +C6 = 1;   +C9 +C10 +C11 +C12 = 1;
Actual optimized value
x[0]=1.00, x[1]=0.00, x[2]=0.000, x[3]=0.000, x[4]=0.500, x[5]=0.500, x[6]=0.000, x[7]=0.00, x[8]=0.00, x[9]=0.00, x[10]=0.428571, x[11]=0.571429,
xl value = 4.000000
Allotted values in iter 1 : x[0] =1 , x[5] =1 , x[11] =1 , : corresponding cost is 2569.000000
Allotted values in iter 2 : x[0] =1 , x[4] =1 , x[11] =1 , : corresponding cost is 2577.000000
Allotted values in iter 3 : x[0] =1 , x[4] =1 , x[11] =1 , : corresponding cost is 2577.000000
Allotted values in iter 4 : x[0] =1 , x[4] =1 , x[11] =1 , : corresponding cost is 2577.000000
Allotted values in iter 5 : x[0] =1 , x[4] =1 , x[10] =1 , : corresponding cost is 2514.000000
Allotted values in iter 6 : x[0] =1 , x[5] =1 , x[10] =1 , : corresponding cost is 2506.000000
Allotted values in iter 7 : x[0] =1 , x[4] =1 , x[11] =1 , : corresponding cost is 2577.000000
Allotted values in iter 8 : x[0] =1 , x[4] =1 , x[11] =1 , : corresponding cost is 2577.000000
Allotted values in iter 9 : x[0] =1 , x[4] =1 , x[10] =1 , : corresponding cost is 2514.000000
Allotted values in iter 10 : x[0] =1 , x[5] =1 , x[10] =1 , : corresponding cost is 2506.000000

Best values are
x[net: 1][tree: 1] = actual= 1.00, alloted = 1   x[net: 1][tree: 2] = actual= 0.00, alloted = 0
x[net: 1][tree: 3] = actual= 0.00, alloted = 0   x[net: 2][tree: 1] = actual= 0.50, alloted = 0
x[net: 2][tree: 2] = actual= 0.500, alloted = 1   x[net: 3][tree: 1] = actual= 0.00 , alloted = 0
x[net: 3][tree: 2] = actual= 0.00, alloted = 0   x[net: 3][tree: 3] = actual= 0.428571 , alloted = 1
x[net: 3][tree: 4] = actual= 0.571429 , alloted = 0
Lower bound on cost = 1746.000000,   **Actual cost after allotment= 2506.000000**

10.8 Left Edge Algorithm

Implementation file is saved as *final_left_edge.c*
How many columns ? (eg: 12) : 12
How many nets ? (eg: 10) : 10
Enter Top net list one by one (eg: 0 1 4 5 1 6 7 0 4 9 10 10) : 0 1 4 5 1 6 7 0 4 9 10 10
Enter bottom net list one by one (eg: 2 3 5 3 5 2 6 8 9 8 7 9) : 2 3 5 3 5 2 6 8 9 8 7 9

After the nets are arranged from left to right,
Ordered nets[2] : start col: 0 , End col: 5
Ordered nets[1] : start col: 1 , End col: 4
Ordered nets[3] : start col: 1 , End col: 3
Ordered nets[4] : start col: 2 , End col: 8
Ordered nets[5] : start col: 2 , End col: 4
Ordered nets[6] : start col: 5 , End col: 6
Ordered nets[7] : start col: 6 , End col: 10
Ordered nets[8] : start col: 7 , End col: 9
Ordered nets[9] : start col: 8 , End col: 11
Ordered nets[10] : start col: 10 , End col: 11
Max density is 5

Edges of VCG GRAPH
v w is 1 5
v w is 1 3
v w is 4 9
v w is 4 5

v w is 5 3
v w is 6 2
v w is 7 6
v w is 9 8
v w is 10 9
v w is 10 7

OUTPUT
track of 1 is 5
track of 2 is 1
track of 3 is 2
track of 4 is 4
track of 5 is 3
track of 6 is 2
track of 7 is 3
track of 8 is 1
track of 9 is 2
track of 10 is 5

10.9 Detailed Routing of Standard Cell with Side Channel

Implementation file is saved as *routing_standard_cell_side_channel_with_left_edge.c*
No need to input any additional values.
Output:
Top Terminals of channel 1 are: 3, 4, 0, 4, 2, 0, 0
Bottom Terminals of channel 1 are: 2, 4, 4, 3, 0, 6, 0
side terminals of channel 1 are : 6, 2, 3
Track for channel 1 are
track of net 3 is 2
track of net 4 is 1
track of net 2 is 3
track of net 6 is 1

Top Terminals of channel 2 are: 0, 7, 1, 6, 5, 1, 0
Since cyclic vertical constraint exists between 7 & 6, shifting cells of top terminal towards right.
Top Terminals of channel 2 after shifting are: 0, 0, 7, 1, 6, 5, 1
Bottom Terminals of channel 2 are: 5, 6, 0, 7, 0, 1, 0
side terminals of channel 2 are : 6
Track for channel 2 are
track of net 7 is 4
track of net 1 is 3
track of net 6 is 2
track of net 5 is 1

Top Terminals of channel 3 are: 0, 2, 6, 3, 0, 0, 0
Bottom Terminals of channel 3 are: 6, 3, 2, 0, 3, 0, 0
side terminals of channel 3 are : 2, 6, 3
Track for channel 3 are
track of net 2 is 2
track of net 6 is 1

track of net 3 is 3

Top Terminals of channel 4 are: 8, 6, 3, 8, 0, 0, 0
Bottom Terminals of channel 4 are: 3, 0, 6, 0, 8, 0, 0
side terminals of channel 4 are :6, 3
Track for channel 4 are
track of net 8 is 1
track of net 6 is 3
track of net 3 is 2

Top Terminals of side channel are: 6, 2, 3, 6, 2, 6, 3, 6, 3
Track for side channel are
track of net 6 is 1
track of net 2 is 2
track of net 3 is 3

10.10 Calculation of Buffer Insertion Segments in the critical path

Implementation file is saved as *buffer_insertion_segments_with_flow_and_mod_Dijkstra.c*
Reading the input values from the file, *graph_data1*

Edges of input Graph from the file are :
start node, end node, weight $\Longrightarrow$ 0, 10, 0
start node, end node, weight $\Longrightarrow$ 0, 6, 0
start node, end node, weight $\Longrightarrow$ 0, 1, 0
start node, end node, weight $\Longrightarrow$ 1, 11, 0
start node, end node, weight $\Longrightarrow$ 1, 7, 0
start node, end node, weight $\Longrightarrow$ 1, 2, 0
start node, end node, weight $\Longrightarrow$ 2, 13, 0
start node, end node, weight $\Longrightarrow$ 2, 3, 0
start node, end node, weight $\Longrightarrow$ 3, 4, 0
start node, end node, weight $\Longrightarrow$ 3, 9, 0
start node, end node, weight $\Longrightarrow$ 4, 5, 0
start node, end node, weight $\Longrightarrow$ 6, 2, 0
start node, end node, weight $\Longrightarrow$ 7, 8, 0
start node, end node, weight $\Longrightarrow$ 8, 3, 0
start node, end node, weight $\Longrightarrow$ 9, 4, 0
start node, end node, weight $\Longrightarrow$ 9, 5, 0
start node, end node, weight $\Longrightarrow$ 10, 12, 0
start node, end node, weight $\Longrightarrow$ 10, 2, 0
start node, end node, weight $\Longrightarrow$ 11, 3, 0
start node, end node, weight $\Longrightarrow$ 12, 4, 0
start node, end node, weight $\Longrightarrow$ 13, 3, 0

Critical path from **node-0** to **node-5** with number of edges as **5**

Output:
For each critical segment, flow graph is constructing from the input graph.
Cost of each segment is the maximum flow value corresponding to its flow graph.

Edges in critical path with cost of cutting as weight are:
start node, end node, cost $\Longrightarrow$ 0, 1, 3
start node, end node, cost $\Longrightarrow$ 1, 2, 5
start node, end node, cost $\Longrightarrow$ 2, 3, 5
start node, end node, cost $\Longrightarrow$ 3, 4, 3
start node, end node, cost $\Longrightarrow$ 4, 5, 2

Modifying the critical path graph by adding red edges of zero weight from all vertices in the critical path to a distance between $minD$-$maxD$.
Splitting each nodes into two, as v and $v + node\ size$. Original blue edges from $v + node\ size$ to w and red edges from v to $w + node\ size$.
start node, end node, cost $\Longrightarrow$ 0 9 0
start node, end node, cost $\Longrightarrow$ 0 8 0
start node, end node, cost $\Longrightarrow$ 1 10 0
start node, end node, cost $\Longrightarrow$ 1 9 0
start node, end node, cost $\Longrightarrow$ 2 11 0
start node, end node, cost $\Longrightarrow$ 2 10 0
start node, end node, cost $\Longrightarrow$ 3 11 0
start node, end node, cost $\Longrightarrow$ 6 1 3
start node, end node, cost $\Longrightarrow$ 7 2 5
start node, end node, cost $\Longrightarrow$ 8 3 5
start node, end node, cost $\Longrightarrow$ 9 4 3
start node, end node, cost $\Longrightarrow$ 10 5 2

Performing Dijkstra's algorithm on modified graph to calculate shortest path from source to target.

Nodes in shortest path from source to target node, are : {6, 1, 10, 5}

Blue edges in the path are: {Edge from 6 to 1, Edge from 10 to 5}

Corresponding edges in the original graph, where buffers to be inserted are:
{Edge from 0 to 1, Edge from 4 to 5}

10.11 Example of Partitioning a Circuit with New Approach

Implementation file is saved as *flip_flop_circuit_partition_sim_annealing.c*
Reading the input values from the file, *many_flip_flop_data2*
Input: Edges of original Graph
start node, end node $\Longrightarrow$ 0, 3
start node, end node $\Longrightarrow$ 0, 7
start node, end node $\Longrightarrow$ 1, 8
start node, end node $\Longrightarrow$ 1, 3
start node, end node $\Longrightarrow$ 2, 4
start node, end node $\Longrightarrow$ 3, 5
start node, end node $\Longrightarrow$ 3, 4
start node, end node $\Longrightarrow$ 4, 10
start node, end node $\Longrightarrow$ 4, 9
start node, end node $\Longrightarrow$ 4, 8
start node, end node $\Longrightarrow$ 4, 6

start node, end node $\Longrightarrow$ 4, 0
start node, end node $\Longrightarrow$ 6, 12
start node, end node $\Longrightarrow$ 6, 5
start node, end node $\Longrightarrow$ 7, 2
start node, end node $\Longrightarrow$ 8, 6
start node, end node $\Longrightarrow$ 8, 5
start node, end node $\Longrightarrow$ 9, 11
start node, end node $\Longrightarrow$ 9, 7
start node, end node $\Longrightarrow$ 9, 5
start node, end node $\Longrightarrow$ 10, 5
start node, end node $\Longrightarrow$ 11, 5
start node, end node $\Longrightarrow$ 12, 9

list of primary input nodes = {1}, list of primary output nodes = {5},
list of flip-flop nodes = {2, 3, 9, 10}.

Output:
Edges of acyclic Graph, which is obtained by converting each flip-flops to two nodes, secondary input and output are:
start node, end node $\Longrightarrow$ 0, 31
start node, end node $\Longrightarrow$ 0, 7
start node, end node $\Longrightarrow$ 1, 8
start node, end node $\Longrightarrow$ 1, 31
start node, end node $\Longrightarrow$ 4, 101
start node, end node $\Longrightarrow$ 4, 91
start node, end node $\Longrightarrow$ 4, 8
start node, end node $\Longrightarrow$ 4, 6
start node, end node $\Longrightarrow$ 4, 0
start node, end node $\Longrightarrow$ 6, 12
start node, end node $\Longrightarrow$ 6, 5
start node, end node $\Longrightarrow$ 7, 21
start node, end node $\Longrightarrow$ 8, 6
start node, end node $\Longrightarrow$ 8, 5
start node, end node $\Longrightarrow$ 11, 5
start node, end node $\Longrightarrow$ 12, 91
start node, end node $\Longrightarrow$ 20, 4
start node, end node $\Longrightarrow$ 30, 5
start node, end node $\Longrightarrow$ 30, 4
start node, end node $\Longrightarrow$ 90, 11
start node, end node $\Longrightarrow$ 90, 7
start node, end node $\Longrightarrow$ 90, 5
start node, end node $\Longrightarrow$ 100, 5

Secondary input and output corresponding to flip flop 2 are 20, 21.
Secondary input and output corresponding to flip flop 3 are 30, 31.
Secondary input and output corresponding to flip flop 9 are 90, 91.
Secondary input and output corresponding to flip flop 10 are 100, 101.

Display of all paths
. . . .

path 0 is $\Longrightarrow$ 1, 31,
path 1 is $\Longrightarrow$ 1, 8, 5
path 2 is $\Longrightarrow$ 1, 8, 6, 5
path 3 is $\Longrightarrow$ 1, 8, 6, 12, 91
path 4 is $\Longrightarrow$ 20, 4, 0, 7, 21
path 5 is $\Longrightarrow$ 20, 4, 0, 31,
path 6 is $\Longrightarrow$ 20, 4, 6, 5,
path 7 is $\Longrightarrow$ 20, 4, 6, 12, 91
path 8 is $\Longrightarrow$ 20, 4, 8, 5
path 9 is $\Longrightarrow$ 20, 4, 8, 6, 5
path 10 is $\Longrightarrow$ 20, 4, 8, 6, 12, 91
path 11 is $\Longrightarrow$ 20, 4, 91,
path 12 is $\Longrightarrow$ 20, 4, 101,
path 13 is $\Longrightarrow$ 30, 4, 0, 7, 21
path 14 is $\Longrightarrow$ 30, 4, 0, 31,
path 15 is $\Longrightarrow$ 30, 4, 6, 5
path 16 is $\Longrightarrow$ 30, 4, 6, 12, 91
path 17 is $\Longrightarrow$ 30, 4, 8, 5
path 18 is $\Longrightarrow$ 30, 4, 8, 6, 5
path 19 is $\Longrightarrow$ 30, 4, 8, 6, 12, 91
path 20 is $\Longrightarrow$ 30, 4, 91
path 21 is $\Longrightarrow$ 30, 4, 101
path 22 is $\Longrightarrow$ 30, 5,
path 23 is $\Longrightarrow$ 90, 5
path 24 is $\Longrightarrow$ 90, 7, 21
path 25 is $\Longrightarrow$ 90, 11, 5
path 26 is $\Longrightarrow$ 100, 5
. . . .

A table is prepared, which shows the nodes connected to each flip-flops.

Matrix which shows the number of nodes common to each pair of flip-flops:

0	7	6	2
7	0	7	2
6	7	0	2
2	2	2	0

Matrix which shows the amount of direct connectivity between the flip-flop pairs:

0	2	2	1
2	0	1	1
2	1	0	0
1	1	0	0

Cost of partitioning of flip flop pairs calculated, from the above two matrices by, first matrix*100 + second matrix*25, are:

0	750	650	225
750	0	725	225
650	725	0	200
225	225	200	0

Edges of undirected flipflop Graph which shows the cost of partitioning of flip-flops are:
start node, end node, cost $\Longrightarrow$ 2, 10, 225
start node, end node, cost $\Longrightarrow$ 2, 9, 650
start node, end node, cost $\Longrightarrow$ 2, 3, 750
start node, end node, cost $\Longrightarrow$ 3, 10, 225
start node, end node, cost $\Longrightarrow$ 3, 9, 725
start node, end node, cost $\Longrightarrow$ 9, 10, 200

By using simulated annealing, the flip-flops to be clustered are:

Group-1 : {flip-flop 9, flip-flop 2, flip-flop 3}

Group-2 : {flip-flop 10}

10.12 Clustering by Simulated Annealing, r-balanced Flow Based Method and Branch and Bound Method

Implementation is saved as, *final_cluster_by_sim_annealing.c*, *final_r_balanced_clustering_Ford_Fulk_flow.c* and *final_branch_and_bound.c*
Input is reading from the file *cluster_data3*
Input
Original Graph is
v w wt is 0 9 23 0
v w wt is 0 8 9 0
v w wt is 0 7 12 0
v w wt is 0 6 15 0
v w wt is 0 5 15 0
v w wt is 0 4 40 0
v w wt is 0 3 30 0
v w wt is 0 2 15 0
v w wt is 0 1 1 0
v w wt is 1 9 25 0
v w wt is 1 8 10 0
v w wt is 1 7 9 0
v w wt is 1 6 7 0
v w wt is 1 2 10 0
v w wt is 1 3 2 0
v w wt is 1 5 8 0
v w wt is 1 4 5 0
v w wt is 2 9 13 0
v w wt is 2 8 21 0
v w wt is 2 7 14 0
v w wt is 2 6 12 0
v w wt is 2 3 15 0
v w wt is 2 5 13 0
v w wt is 2 4 12 0
v w wt is 3 9 27 0
v w wt is 3 8 12 0
v w wt is 3 7 21 0

v w wt is 3 6 18 0
v w wt is 3 5 12 0
v w wt is 3 4 19 0
v w wt is 4 9 32 0
v w wt is 4 8 7 0
v w wt is 4 7 15 0
v w wt is 4 6 14 0
v w wt is 4 5 60 0
v w wt is 5 9 10 0
v w wt is 5 8 31 0
v w wt is 5 7 23 0
v w wt is 5 6 12 0
v w wt is 6 9 16 0
v w wt is 6 8 140 0
v w wt is 6 7 11 0
v w wt is 7 9 17 0
v w wt is 7 8 23 0
v w wt is 8 9 20 0

Result from Simulated Annealing

Initial set is : 0 1 2 3 4 5 6 7
Current set is : 7 , 1 , 2 , 3 , 4 , 5 , 6 , 0 ,
Current set is : 7 , 8 , 2 , 3 , 4 , 5 , 6 , 0 ,
Current set is : 7 , 8 , 9 , 3 , 4 , 5 , 6 , 0 ,
Current set is : 7 , 8 , 9 , 0 , 4 , 5 , 6 , 3 ,
Current set is : 8 , 0 , 9 , 2 , 4 , 6 , 7 , 5 ,
Current set is : 6 , 4 , 7 , 8 , 0 , 9 , 5 , 3 ,
Current set is : 7 , 4 , 8 , 0 , 5 , 3 , 6 , 2 ,
Current set is : 7 , 4 , 8 , 0 , 2 , 3 , 6 , 5 ,
Current set is : 3 , 6 , 5 , 7 , 4 , 8 , 0 , 9 ,
Current set is : 0 , 6 , 5 , 7 , 4 , 8 , 3 ,
Current set is : 7 , 5 , 8 , 4 , 0 , 6 , 9 ,
Current set is : 7 , 5 , 8 , 4 , 0 , 9 , 6 ,
Current set is : 4 , 0 , 9 , 6 , 7 , 5 , 8 ,
Current set is : 4 , 0 , 9 , 8 , 7 , 5 , 6 ,
Current set is : 4 , 0 , 9 , 8 , 2 , 5 , 6 ,
Current set is : 4 , 0 , 9 , 8 , 2 , 3 , 6 ,
Current set is : 8 , 7 , 5 , 6 , 4 , 0 , 9 ,
Current set is : 8 , 9 , 5 , 6 , 4 , 0 , 7 ,
Current set is : 6 , 4 , 5 , 0 , 8 , 7 , 3 ,
Current set is : 6 , 4 , 5 , 0 , 8 , 9 , 3 ,
Current set is : 6 , 4 , 5 , 0 , 8 , 9 , 3 ,
Current set is : 3 , 7 , 4 , 6 , 5 , 8 , 9 ,
Current set is : 8 , 7 , 4 , 6 , 5 , 3 ,
Current set is : 8 , 9 , 4 , 6 , 5 , 3 ,
Current set is : 7 , 9 , 1 , 0 , 2 , 3 ,
Current set is : 7 , 4 , 6 , 5 , 3 , 8 ,
Current set is : 1 , 9 , 8 , 3 , 5 , 6 ,
Current set is : 1 , 9 , 8 , 2 , 5 , 6 ,
Current set is : 1 , 9 , 8 , 2 , 0 , 6 ,

Current set is : 4 , 7 , 3 , 5 , 0 , 1 ,
Current set is : 9 , 8 , 2 , 5 , 6 , 1 ,
Current set is : 9 8 , 2 , 0 , 6 , 1 ,
Current set is : 9 , 8 , 2 , 0 , 1 , 6 ,
Current set is : 7 , 3 , 5 , 0 , 1 , 4 ,
Current set is : 7 , 3 , 5 , 4 , 1 , 0 ,
Current set is : 8 , 3 , 5 , 4 , 6 , 9 ,
Current set is : 8 , 2 , 9 , 4 , 6 , 5 ,
Current set is : 8 , 2 , 9 , 1 , 6 , 5 ,
Current set is : 3 , 2 , 9 , 1 , 0 , 7 ,
Current set is : 3 , 7 , 9 , 1 , 0 , 2 ,
Current set is : 3 , 7 , 4 , 6 , 5 , 8 ,
Current set is : 5 , 8 , 4 , 6 , 3 ,
Current set is : 8 , 5 , 6 , 9 , 1 ,
Current set is : 8 , 2 , 6 , 9 , 1 ,
Current set is : 1 , 7 , 2 , 0 , 9 ,
Current set is : 1 , 7 , 3 , 0 , 9 ,
Current set is : 4 , 6 , 8 , 5 , 7 ,
Current set is : 8 , 6 , 1 ,
Current set is : 6 , 1 , 8 ,
Current set is : 3 , 0 , 1 ,
Current set is : 3 , 0 , 7 ,
Current set is : 9 , 0 , 7 ,
Current set is : 0 , 7 , 9 , 4 ,
Current set is : 5 , 6 , 8 , 2 ,
Current set is : 3 , 0 , 9 , 1 ,
Current set is : 0 , 9 , 1 , 7 ,
Current set is : 1 , 9 , 0 ,
Current set is : 1 , 7 , 0 ,
Current set is : 9 , 0 , 1 ,
Current set is : 0 , 1 , 9 ,
Current set is : 8 , 6 , 3 , 0 ,
Current set is : 8 , 6 , 2 , 0 ,
Current set is : 2 , 8 , 6 ,
Current set is : 1 , 9 , 2 ,
Current set is : 2 , 9 , 1 ,
Current set is : 9 , 1 , 2 ,
Current set is : 1 , 2 , 9 ,
Current set is : 2 , 7 , 1 ,
Current set is : 9 , 1 , 2 ,
Current set is : 7 , 1 , 2 ,
Current set is : 1 , 2 , 7 ,
Current set is : 7 , 2 , 1 ,
Current set is : 1 , 7 , 2 ,
Current set is : 7 , 2 , 1 ,
Final clustered set is :
Group 1 : 7 2 1
Group 2 : 4 9 0 6 5 8 3
Group-1 edge wt is 33 and Group-2 edge wt is 562
Cutting edge weight is 281

Result from r balanced flow based method

Enter the epsilon value as 0 to 0.9 (Lower value will take more time) : 0.2

Performed the maximizing flow algorithm and calculated the minimum cut
Initial set of Source side nodes corresponding to the FLOW graph are : [0 2 11 20 21 22 23 24 25 26 27]
Initial set of Source side nodes corresponding to ORIGINAL graph are : [1]

(1-epsilon)rW = 3 and (1+epsilon)rW = 6

Performed the flow algorithm and calculated the minimum cut
Since, number of source side nodes of ORIGINAL graph <(1-epsilon)rW, All source side nodes of FLOW graph is to be clustered

Enter the extra node to be clustered with the SOURCE from the given flow node list: [10 9 8 7 6 5 4 3 1]: 3
Extra node clustered with the source is 3
Source clustered nodes corresponding to FLOW graph are : [0 2 3 11 20 21 22 23 24 25 26 27]
Target clustered nodes corresponding to FLOW graph are : [56]

Performed the flow algorithm and calculated the minimum cut
New Source side nodes corresponding to the FLOW graph are : [0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55]
New Source side nodes corresponding to ORIGINAL graph are : [0 1 2 3 4 5 6 7 8 9]

Since, number of source side nodes of ORIGINAL graph > (1+epsilon)rW, cluster all target side nodes of FLOW graph

Enter the extra node to be clustered with the TARGET from the given FLOW node list: [12 13 14 15 16 17 18 19 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55]:55
Extra node clustered to the Target is 55
Source clustered nodes corresponding to FLOW graph are : [0 2 3 11 20 21 22 23 24 25 26 27]
Target clustered nodes corresponding to FLOW graph are : [55 56]

Performed the flow algorithm and calculated the minimum cut
New Source side nodes corresponding to the FLOW graph are : [0 2 3 11 12 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34]
New Source side nodes corresponding to ORIGINAL graph are : [1 2]

Since, number of source side nodes of ORIGINAL graph < (1-epsilon)rW, All source side nodes of FLOW graph is to be clustered

Enter the extra node to be clustered with the SOURCE from the given flow node list: [1 4 5 6 7 8 9 10]: 8
Extra node clustered with the source is 8
Source clustered nodes corresponding to FLOW graph are : [0 2 3 8 11 12 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34]
Target clustered nodes corresponding to FLOW graph are : [55 56]

Performed the flow algorithm and calculated the minimum cut
New Source side nodes corresponding to the FLOW graph are : [0 2 3 8 11 12 17 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 38 43 47 50 53 54]
New Source side nodes corresponding to ORIGINAL graph are : [1 2 7]

Final Source side nodes corresponding to ORIGINAL graph are : [1 2 7]
Final Target side nodes corresponding to ORIGINAL graph are : [0 3 4 5 6 8 9]

Source side wt is 33 and Target side wt is 562
Cutting edge weight is 281

Result from Branch and Bound method

Initial Solution $\Longrightarrow$ SET-1 = [0 1 2 3 4 5 9], SET-2 = [6 7 8]
Minimum cutting edge wt = 315, Size of set-1= 7, Corresponding COST = 2493

Better Solution $\Longrightarrow$ SET-1 = [0 1 2 3 4 5], SET-2 = [6 7 8 9]
Minimum cutting edge wt = 392, Size of set-1= 6, Corresponding COST = 2392

Better Solution $\Longrightarrow$ SET-1 = [0 1 2 3 4 9], SET-2 = [5 6 7 8]
Minimum cutting edge wt = 367, Size of set-1= 6, Corresponding COST = 2367

Better Solution $\Longrightarrow$ SET-1 = [0 1 2 3 4], SET-2 = [5 6 7 8 9]
Minimum cutting edge wt = 424, Size of set-1= 5, Corresponding COST = 2274

Better Solution $\Longrightarrow$ SET-1 = [0 1 2 3 7], SET-2 = [4 5 6 8 9]
Minimum cutting edge wt = 405, Size of set-1= 5, Corresponding COST = 2255

Better Solution $\Longrightarrow$ SET-1 = [0 1 2 3 9], SET-2 = [4 5 6 7 8]
Minimum cutting edge wt = 379, Size of set-1= 5, Corresponding COST = 2229

Better Solution $\Longrightarrow$ SET-1 = [0 1 2 3], SET-2 = [4 5 6 7 8 9]
Minimum cutting edge wt = 372, Size of set-1= 4, Corresponding COST = 2122

Better Solution $\Longrightarrow$ SET-1 = [0 1 2 9], SET-2 = [3 4 5 6 7 8]
Minimum cutting edge wt = 371, Size of set-1= 4, Corresponding COST = 2121

Better Solution $\Longrightarrow$ SET-1 = [0 1 2], SET-2 = [3 4 5 6 7 8 9]
Minimum cutting edge wt = 310, Size of set-1= 3, Corresponding COST = 2060

Better Solution $\Longrightarrow$ SET-1 = [1 2 3], SET-2 = [0 4 5 6 7 8 9]
Minimum cutting edge wt = 304, Size of set-1= 3, Corresponding COST = 2054

Better Solution $\Longrightarrow$ SET-1 = [1 2 7], SET-2 = [0 3 4 5 6 8 9]
Minimum cutting edge wt = 281, Size of set-1= 3, Corresponding COST = 2031

Total number of searched (feasible) solutions = 840 out of 1024 solutions

Best Solution with Cutting Edge Weight = 281.
SET-1 = [1 2 7], SET-2 = [0 3 4 5 6 8 9]

Bibliography

[1] M. Sarrfzadeh, C. K. Wong, *An introduction to physical design*, The McGraw-Hill Companies, Inc., 1996.

[2] N. Sherwani,*Algorithms for VLSI physical design, Automation*, Kluwer Academic Publishers, 2002.

[3] M. Michael Vai, *VLSI Design*, CRC press, Indian reprint in 2009.

[4] Thomas Lengauer, *Combinatorial Algorithms for Integrated Circuit Layout*, John Wiley and Sons, 1990.

[5] Mazumder P., E. M. Rudnick,*Genetic Algorithm for VLSI Design, Layout and Test Automation*, Prentice Hall , 1999.

[6] Robert Sedgewick,*Algorithms in C, Third Edition,*, Addison-Wesley, 1998.

[7] Thomas H.Cormen, E. Leiserson, L.Rivest, C. Stein, *Introduction To Algorithms, Third Edition,* PHI Learning Private Limited, 2010.

[8] S. Sahni, *Data Structures, Algorithms, and Applications in C++,* McGraw Hill, NY, second edition, silicon press ed., 2005.

[9] P. H.Narayanan and P. S. Patkar, *Handbook of Graph Theory and Algorithms*. 1st ed.,2010

[10] Abhijit Shripad Deshpande,*VLSI Circuit Partitioning*,Mtech project report IIT Bombay, 2005.

[11] H. Yang and D.F. Wong, *Efficient Network Flow Based Min-Cut Balanced Partitioning*,IEEE Trans. On CAD Integrated Circuits and Systems, vol. 15, no. 12, 1996, pp. 1533-1540.

[12] Kun-Mao Chao, Bang Ye Wu,*Steiner Minimal Trees* ,An excerpt from the book Spanning Trees and Optimization Problems, by Bang Ye Wu and Kun-Mao Chao (2004), Chapman and Hall/CRC Press, USA.

[13] P. Raghavan,C. D. Thompson,*Randomized routing: A technique for provably good algorithms and algorithmic proofs*,Combinatorica 7(4) (1987) 365-377.

[14] Chung Kuan Cheng,E. S. Kuh,*Module Placement Based on Resistive Network Optimization*, IEEE Transactions on Computer Aided Design, VOL. CAD-3, NO.3, JULY 1984.

[15] David B. Shmoys,Eva Tardos, *An approximation algorithm for the generalized assignment problem*,Mathematical Programming 62 (1993) 461-474

[16] Minghorng Lai, D. F. Wong, *Slicing Tree Is a Complete Floorplan Representation*, Department of Computer Sciences, The University of Texas at Austin, Austin, TX 78712, USA.

[17] Gary Kok-Hoo Yeap,*A Unified Approach to Floorplan Sizing and Enumeration*, IEEE Transactions on Computer Aided Design, VOL.12 , NO.12, JULY 1993.

[18] Daniel Cordeiro,*Random graph generation for scheduling simulations*,Operational research of the CNRS , 1999.

[19] Farhana Johar, Shaharuddin Salleh,*Placement and Routing in VLSI design Problem Using Single Row Routing Technique*,MATEMATIKA, Volume 23, Number 2, 99120,2007.

[20] http://en.wikipedia.org/

[21] http://lpsolve.sourceforge.net/5.5/

[22] http:www.scilab.org