Notes by Mehul Shah

- Introduction to VLSI Routing
- Global Routing
- Two terminal routing (shortest paths)
  1. Dijkstra's algorithm
  2. Maze routing
- Multi terminal routing (Steiner trees)
  1. Steiner tree definitions
  2. Optimal rectilinear steiner trees

1 Introduction to VLSI Routing

In the physical design of VLSI circuits, the *routing* phase consists of finding the layouts for the wires connecting the terminals on circuit blocks or gates. It assumes the locations of the blocks have already been determined, although some routing schemes allow a certain amount of movement of the blocks to make space for wires. The basic problem is illustrated in Figure 122. A *terminal* is a location on a block, typically along one of the edges, which needs to be connected to other blocks. A *net* is a set of terminals that all need to be connected to each other. A *netlist* is a collection of nets that need to be routed. The *routing space* is the space in which we are allowed to place wires, typically any space not taken by blocks. The *routing space* between two blocks is called a *channel*. The routing problem is the problem of locating a set of wires in the routing space that connect all the nets in the netlist. The capacities of channels, width of wires, and wire crossings often need to be taken into consideration in the problem.

Given that there can be millions of transistors on a chip and tens of thousands of nets to route, each with a large number of possible routes, the routing problem is computationally very difficult. In fact, as we will see, even finding an optimal layout (minimum wire-length) for a single net is NP-hard. Because of the hardness of the problem, most routing algorithms are approximate and will not necessarily find the best solution. The first simplification that is typically made is to divide the problem into two phases called *global routing* and *detailed routing*. In global routing the exact geometric details are ignored and only the "loose route" for the wires are determined. More specifically, the open regions through which each wire flows is computed. This loose route is converted to an exact layout in the latter phase. Detailed routing completes the point-to-point wiring by specifying geometric information such as the location and width of wires and their layer assignments. Even with this breakdown into global and detailed routing each phase remains NP-hard and further

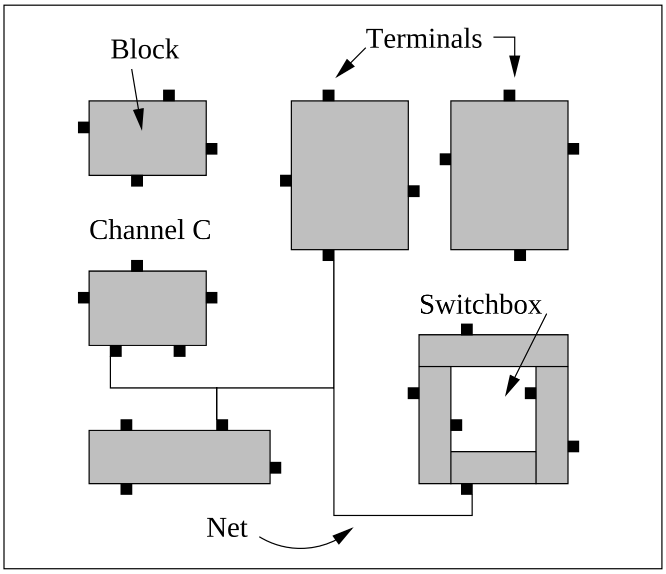


Figure 122: Some definitions.

approximations are used to solve them. In this lecture, we concentrate on the global routing phase.

The optimality goal of global routing is dependent upon the nature of the chip design. For full custom and standard cell designs, the objective is to minimize the total area of the final chip. Thus, in the full custom designs the total wire length is minimized, or the maximum wire length is minimized, or changes to placement are minimized. In the standard cell design, the maximum wire length is minimized or the total channel heights are minimized. In gate arrays, the previous phases assign gates to blocks, and the primary goal is to determine the feasibility of routing. If feasible, the maximum wire lengths are minimized; otherwise, different assignments are sought. Variants of shortest paths algorithms are employed for many of these computations.

2 Global Routing Algorithms

In the global routing phase, the input is a netlist consisting of a set of nets each of which is a set of terminals. We will distinguish between two-terminal nets and multi-terminal nets. Most chips will have both types and different algorithms are often used for each.

In addition to the netlist we need some way to specify the routing space. This is typically done as a graph. There are two graph models which are often used: the grid graph model and the channel-intersection graph model. In the *grid graph* model the chip is modeled as a regular $n \times m$ grid graph with vertices that do not belong to the routing space removed (*e.g.*, vertices occupied by a block, or possibly already routed to capacity). Wires can be routed along any edges of this graph and one typically assumes each edge has unit capacity (*i.e*, only one wire can be routed on it). All edges in this model have the same length. Figure 126 shows an example—the shaded vertices belong to blocks and therefore do not belong to the

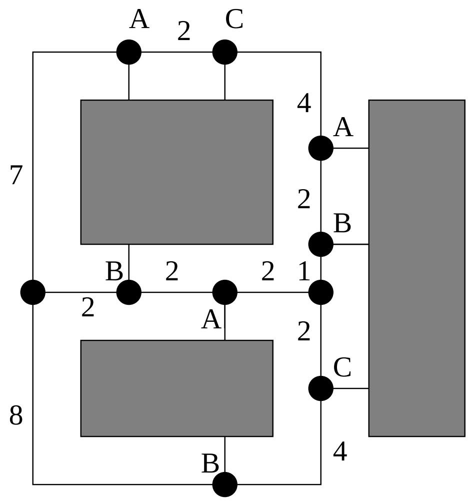


Figure 123: An example of a channel-intersection graph model. The circles are the vertices, the letters are terminals, and the numbers specify the length of each edge.

graph. In the *channel-intersection graph* model vertices are placed at all intersections of the channels, and at the terminals. The lengths of the edges are no longer all the same and the capacity of each edge represents the number of wires that can be routed along it. Figure 123 shows an example. We will consider algorithms for both types of graphs.

Generally, two approaches are taken to route the nets. The first approach is to route sequentially by laying down one net at a time based on constraints from previous nets, and the second is to route all the nets simultaneously. The first approach will typically not give as good routes, but is computationally much more feasible. In this lecture we will just consider the first approach and in the next lecture we will consider an algorithm based on integer-programming for the second approach.

When laying down the nets sequentially, depending on the order the nets are laid out, some nets already routed may block others yet to be routed. One approach to mollify this problem is to sequence the net according to various factors: its criticality, perimeter of the bounding rectangle, and number of terminals. However, sequencing techniques are not always sufficient, in which case, "rip-up and reroute" and "shove-aside" techniques are employed. The "rip-up and reroute" technique involves removing the offending wires which block the affected nets and rerouting them. The "shove-aside" technique moves wires to allow for completion of failed connections. Another approach is to route nets with two or three terminals first since there are few choices for routing them and they comprise a large portion of the netlist.

We now consider the following algorithms for laying down each net.

- Two terminal algorithms (shortest paths)
  1. Dijkstra's algorithm (intersection graphs)
  2. Maze routing and Hadlock's algorithm (grid graphs)
- Multi-terminal algorithms (Steiner trees)

1. Optimal rectilinear steiner trees

3 Two-Terminal Nets: Shortest Paths

The shortest path problem we consider involves an undirected graph $G = (V, E)$ with positive edge weights (representing the length of each edge). Our goal is to find a path from a given source to a destination that minimizes the sum of edges weights along the path.

3.1 Dijkstra's Algorithm

Dijkstra's algorithm is one such single source shortest path algorithm, and in fact in its general form finds the shortest path from the source to all vertices. The algorithm starts from the source vertex and proceeds outward maintaining a set S of vertices that have been searched. Any vertex that is not in S but is a neighbor of a vertex in S is said to be in the *fringe*. It proceeds until all vertices are labeled with their shortest distance from the source (or until the destination is encountered). The algorithm and an example of one of the steps is shown in Figure 124. The last three lines maintain the invariant between iterations since $d[v]$ is correct when v is added to S and because for all new fringe nodes their shortest path through $S \cup \{v\}$ is either through S, which was previously correct, or through v which has length $d[v] + w(u, v)$.

If the fringe vertices are maintained in a priority queue with their distance to S as the priority, then each `findmin` and modification of the $d[u]$ takes $O(\log n)$ time, where $n = |V|$. Since there are at most n `findmin` operations and at most $m = |E|$ modifications to the $d[u]$ (one for each edge), the time is bound by $O(m \log n)$. A Fibonacci heap implementation of the queue, however, can reduce the amortized running time to $O(n \log n + m)$ (see CLR). Note, if all edge weights are constant and equal then this method reduces to breadth-first search.

If all we want to find is the shortest path to a single destination d, then Dijkstra's algorithm can terminate as soon as $d \in S$ (as soon as d is visited). The algorithm, however, moves uniformly in all directions from the source in a best-first manner, so in general a large part of the graph might be visited before d. For graphs which can be embedded in Euclidean space (i.e. the triangle inequality holds for edges within the graph), the efficiency of this search can be improved by biasing the search toward the destination. This optimization can be very beneficial when making two-terminal connections in VLSI layout.

To bias the search toward the destination the modified algorithm uses a different priority for selecting the next vertex to visit. Instead of $d[v]$, the length of the path from the source, it uses $d[v] + L(v, d)$, where $L(v, d)$ is the Euclidean distance from v to the destination d. By including $L(v, d)$, the search will be biased toward vertices with shorter distance to the destination. However, we have to make sure that the algorithm still works correctly. This is guaranteed by the following theorem, which is illustrated Figure 125.

Theorem 2 *For a step of the modified Dijkstra's algorithm, in Figure 125 if $P_1 + L_1 \leq P_2 + L_2$ then $P_1 \leq P_2 + P_3$.*

```
function Dijkstra(G, s)
    // G = (V, E) is a input graph with nonnegative edge weights w(u, v), (u, v) ∈ E
    // s is the source vertex

    // Result: for each v ∈ V, d[v] = shortest distance from s to v

  forall (v ∈ V), d[v] = ∞
  d[s] = 0
  S = ∅
  while S ≠ V // not all vertices have been searched
      // Loop Invariants:
      //    for all v ∈ S, d[v] is shortest path from s to v
      //    for all v ∈ Fringe, d[v] is shortest path exclusively through S from s to v

      v = node v ∈ Fringe with minimum d[v]

      // Note that d[v] is the shortest path to v from s because
      //    1: shortest path through S to v is d[v] (by the loop invariant)
      //    2: any path through a vertex u ∈ (V − S) must be at least as long as d[v]
      //       since d[u] ≥ d[v] and all edge weights are nonnegative.

      S = S ∪ {v} // "visit" v
      for u ∈ neighbors(v)
          d[u] = min(d[u], d[v] + w(v, u))
end
```

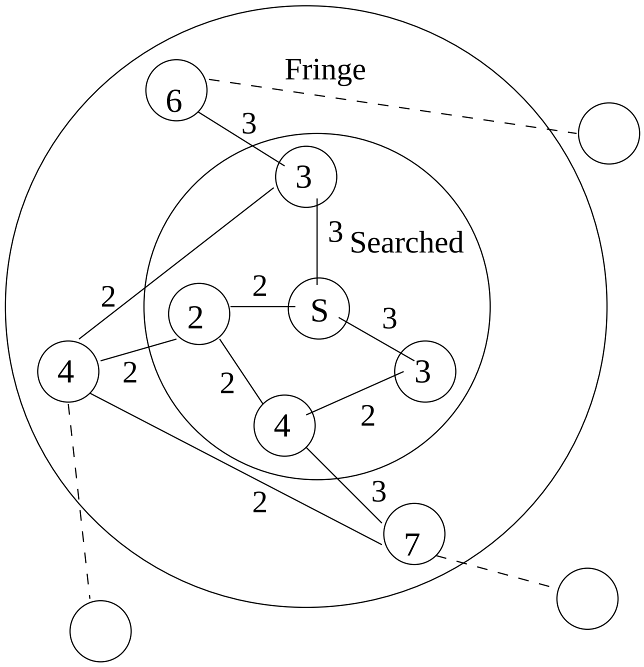


Figure 124: The code for Dijkstra's algorithm and a snapshot of one iteration.

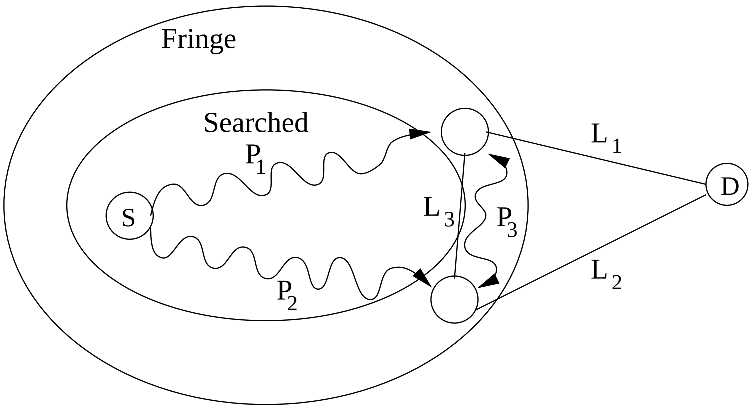


Figure 125: Incorporating the Euclidean distance into the priority maintains the invariant while forcing Dijkstra's algorithm to move in the direction of the destination.

Proof: $P_3 \geq L_3$, a property of Euclidean graphs. Also, $L_3 \geq L_2 - L_1$, which follows from the triangle inequality. Our assumption states:

$$P_1 \leq P_2 + L_2 - L_1$$

Thus,

$$P_1 \leq P_2 + L_3$$
$$P_1 \leq P_2 + P_3$$

□

This theorem implies that if we use $P + L$ as the priority then when $P_1 + L_1$ is the fringe vertex with the minimum priority we are sure there is no shorter path to it through some u not in the searched vertices S and our invariant is maintained.

Exercise: *Consider an $n \times n$ grid graph in which all edges have unit weight. If the source is in the middle of the left boundary and the destination is in the middle of the right boundary, how many vertices will the modified Dijkstra's algorithm take compared to the original? How about if the source is on the lower left and the destination on the upper right?*

3.2 Maze Routing

Finding shortest paths on a grid-graph is a special case of the general shortest path problem where all weights are the same (unit). In the context of VLSI routing, finding such shortest paths for two-terminal nets is called maze routing. A simple approach for maze routing is to perform a breadth-first search from source to destination (this is referred to as "Lee's

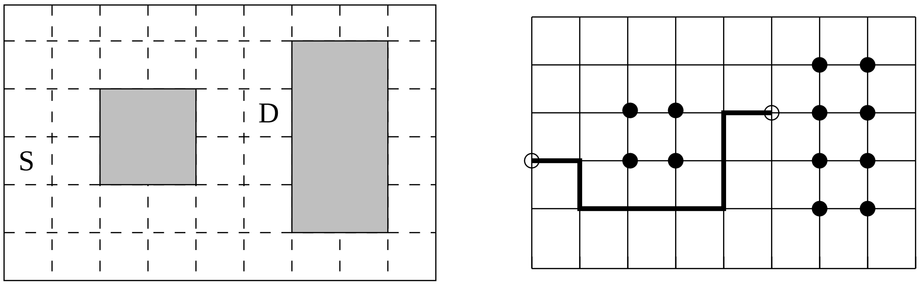


Figure 126: The right figure shows the path determined by a maze route between S and D on a grid graph model of the circuit area in the left figure. For Hadlock's algorithm, $M(S, D) = 6$, and $d(P) = 1$ because the path turns away from D once.

Algorithm" in the VLSI community). Such a search is faster than Dijkstra's algorithm since it only takes constant time to visit each vertex. However, like Dijkstra's unmodified algorithm it makes no attempt at heading toward the destination.

In a similar spirit to the modified version of Dijkstra's algorithm Hadlock observed that the length of a path on a grid from s to t is $M(s, t) + 2d(P)$, where $M(s, t)$ is the Manhattan distance and $d(P)$ is the number of edges on path P directed away from the target (see Figure 126). Hence, the path can be minimized if and only if $d(P)$ is minimized. Thus, instead of labeling each vertex on the fringe with the distance, it is labeled with the number of times the best path to it has turned away from the target. Hadlock's algorithm, therefore uses the detour number $d(P)$ for the priority in the shortest-path search. As with the modified version of Dijkstra's algorithm, this has the effect of moving the search toward the destination.

4 Multi-Terminal Nets: Steiner Trees

4.1 Steiner Trees

Given a graph with a subset of vertices labeled as demand points, a minimum Steiner tree is the minimum set of edges which connect the demand points. Figure 127 gives an example of a minimum Steiner tree. A minimum spanning tree (MST) is a tree which connects all vertices with edges with minimum total weight. Finding the minimum Steiner tree is an NP-hard problem; however, finding the minimum spanning tree is tractable and, in fact, quite efficient. Thus, finding a minimum spanning tree is the initial step in many of the algorithms for approximating minimum Steiner trees.

Usually in VLSI circuits, wires are laid down only in horizontal or vertical directions. Thus, for multi-terminal nets, the demand points are a set of vertices on a grid, and the routing problem is to find a minimum rectilinear Steiner tree (RST), one with horizontal or vertical edges. Finding the minimum RST remains NP-hard so heuristics are employed. A useful fact to note, however, is that the cost of a minimum RST is at most 3/2 the cost of a

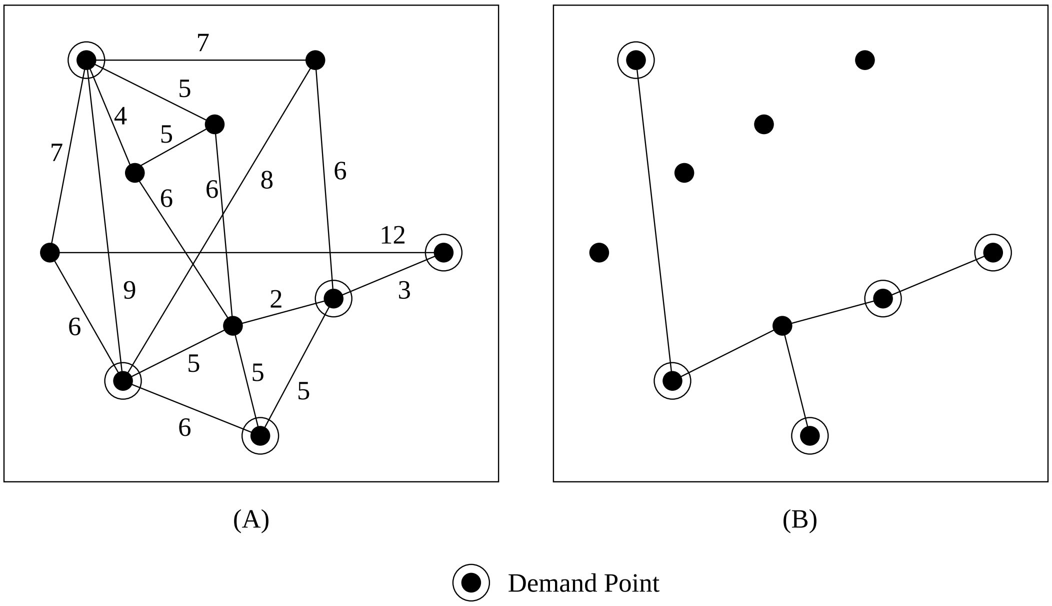


Figure 127: Given the demand points, (B) shows the minimum cost Steiner tree of (A).

minimum spanning tree. Thus, in practice, an approximate minimum RST is computed by rectilinearizing each edge of a minimum spanning tree (see Figure 128).

Various methods for rectilinearizing edges in a MST exist. A layout which transforms each diagonal edge using only one turn on the grid is called an L-RST. A layout which uses two turns for each diagonal edge is called a Z-RST. A layout which uses an arbitrary number of turns but never goes "away" from the destination is called an S-RST. These variants are illustrated in Figure 129. Notice that an optimal RST may have costs smaller than the optimal S-RST, which may have cost smaller than the optimal Z-RST, which likewise may have a cost smaller than the optimal L-RST. The fewer the restrictions, the better the approximation to a minimum RST.

4.2 Optimal S-RST for separable MSTs

If a MST of a given graph obeys the separability property, then there exists a method to find an optimal S-RST from the MST. The separability property states that any pair of non-adjacent edges must have non-overlapping bounding rectangles (see Figure 130). Given such a graph the approximate RST algorithm works as follows. First, it calculates the minimum spanning tree of the given points. Then, the MST is hung by one of the leaves of the tree. Then starting at the root edge it recursively computes the optimal Z-layouts of the subtrees rooted at the children. Each combination of optimal Z-RSTs of the children is merged in with the Z-layout of the root node. The least cost merged layout is the optimal Z-layout of the entire tree. Pseudocode for this algorithm is given in Figure 130 (D).

Note, the separability of the MST insures that the optimal S-RST has the same cost as the optimal Z-RST. This algorithm runs in polynomial time and provides an optimal S-RST, but not necessarily an optimal RST!

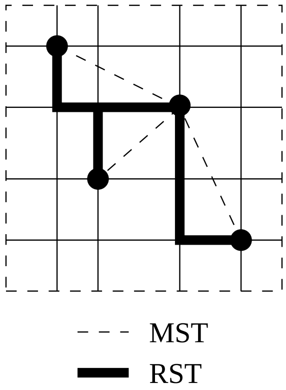


Figure 128: A comparison of a rectilinear Steiner tree (RST) and minimum spanning tree (MST) on a grid graph.

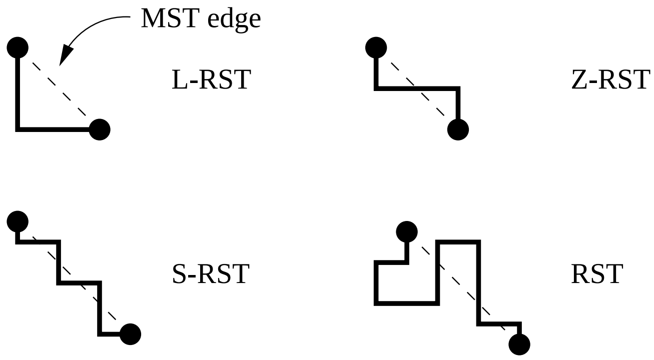
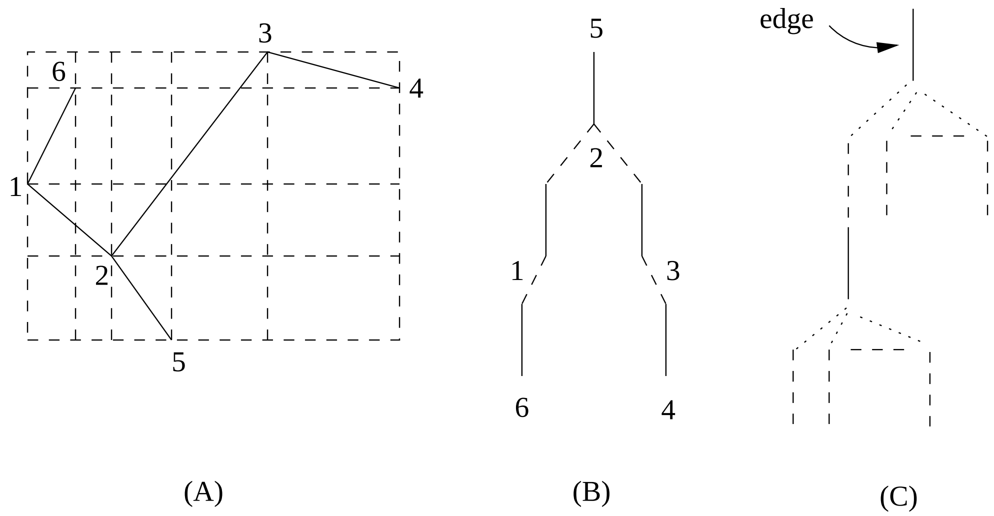


Figure 129: This illustrates the various transformations of an MST edge into rectilinear edges.



function LeastCost(z_e,T_e)
  // z_e a Z-layout for edge e
  // T_e the subtree of edges hanging from e (including e)

  // Returns the minimum cost Z-layout of edges in T_e with the constraint that
  //   edge e is laid out using z_e (all other edges are unconstrained).

begin
  if no children in T_e **then return** z_e
  else
    for each child edge e_i of T_e
      for each Z-layout z_{ij} of e_i
        L_{ij} = LeastCost(z_{ij},T_{e_i})

    for each combination $(L_{1x}, L_{2y}, \ldots)$
      $M_{xy\ldots}$ = Layouts $L_{1x}, L_{2y}, \ldots$ merged with z_e

    return minimum cost $M_{xy\ldots}$
end

(D)

Figure 130: (A) shows a separable MST, (B) shows the MST in (A) hung by leaf 5, (C) shows a generic MST hung by one of its leaves upon which the LeastCost function operates, and (D) shows the code for the LeastCost function.