# RDL Routing report

B04901056 張承洋

Source Code : https://github.com/b04901056/dsa2017/tree/master/rdl_routing

## Task 1 : A-star Search Algorithm Practice

### *A. Problem Formulation*

Given a set of unrouted nets, the objective is to connect these nets through a series of wires. The die area is a rectangular area. All components (pins, wires and obstacles) must be located in this area.

### *B. Algorithm Flow*

A* Search Algorithm picks the node according to a 'F' value at each step. 'F' is a parameter equal to the sum of two other parameters – 'G' and 'H'. At each step it picks the node having the lowest 'F', and process it.

We define 'G' and 'H' below :
G = the wire length to move from the starting pin to a given node on the grid, following the path generated to get there.
H = the estimated wire length to move from that given node on the grid to the final destination pin. This is often referred to as the heuristic.

There are several ways of calculating H value, and we are not sure about which way leads to best solution. However, if we can ensure the H value is always lower than the actual wire length to move from that given node on the grid to the final destination pin, A* algorithm promises the shortest path between two given pins. I choose Manhattan distance for calculating H value.
The figure below shows the detail of implementing A* algorithm.
(source: https://www.youtube.com/watch?v=-L-WgKMFuhE)

```
OPEN //the set of nodes to be evaluated
CLOSED //the set of nodes already evaluated
add the start node to OPEN

loop
 current = node in OPEN with the lowest f_cost
 remove current from OPEN
 add current to CLOSED

 if current is the target node //path has been found
  return

 foreach neighbour of the current node
  if neighbour is not traversable or neighbour is in CLOSED
   skip to the next neighbour

  if new path to neighbour is shorter OR neighbour is not in OPEN
   set f_cost of neighbour
   set parent of neighbour to current
   if neighbour is not in OPEN
    add neighbour to OPEN
```

## C. *Data Structure*

- Node
  Records the coordinate, F G H values, and the parent (used to back trace the wire path after finding the final destination) of a node.
- Net
  Records the net name, source node and destination node of a net.
- Obstacle
  Records the obstacle name, bottom-left and top-right corner point.
- Die area
  After reading in the input file, I construct a two-dimension array to record the status of each node on the grid (wire, obstacle or pin)
- Close set
  I use a two-dimension array to record the Boolean value of each node, and determine whether it has been evaluated or not.
- Open set
  Since we have to find the node in Open_set with lowest F value in each iteration, I choose priority queue and store the Node structure in it.
- Shortest value
  Although priority queue can help us find the node with lowest F value, there may be several elements in Open_set with the same node coordinate (on the grid) but having different F values. The reason for this phenomenon is that there are several different paths from the source node to a specific node.
  Therefore, I maintain a two-dimension array Shortest_value to record the best F value (corresponds to the best path) to each node, neglecting all elements having different F value while finding the node with lowest F value in Open_set.

## D. *Implementation detail*

- Limited routing area
  Given bottom-left and top-right corner points, my router can search for the shortest path within this area.
- X – routing
  In addition to vertical routing, the 45 and 135 degree slant lines are also available for wire connection.

## E. *Result*

My router can find the shortest path between the source node and the

termination node of each net.

- *g++ astar.cpp -o astar*
- *./astar test_astar.txt astar_out.txt*

The result will be in the file 'astar_out.txt', including whether the net is successfully routed or not, and its corresponding wire path.

## Task 2 : The RDL routing problem

### *A. Problem Formulation*

Given a set of unrouted nets, the objective of RDL routing is to connect the chip I/O pads to the corresponding bump pads through several RDL layers with wire length (routing cost) minimized. The relations between chip I/O pads and bump pads are all predefined. That is, each I/O pad must be connected to a specific bump pads.

### *B. Algorithm Flow*

I refer to the paper 'An Efficient Pre-assignment Routing Algorithm for Flip-Chip Designs Chung-Wei Lin, Po-Wei Lee, Yao-Wen Chang, *Senior Member, IEEE,* Chin-Fang Shen, and Wei-Chih Tseng' to solve this problem.

The terms I defined in the following section :

I. IO_connected_pad
   For each I/O pin, IO_connected_pad records the ID of its corresponding bump pad.
II. Pad_string
   Using a cut line, we can transform each ring of bump pads into linearly ordered string structure.
III. LCS_string
   The longest common subsequence of each string in Pad_string and IO_connected_pad
IV. Result_string
   The bump pads string sequence of escape routing on each ring
V. MPSC_priority
   The result of MPSC algorithm in each iteration tells us the maximum set of nets that can be routed prior to others without detour.

Step 1 : String construction

- Since it is hard to tackle all bump pads at the same time, we have to use a cut line to transform the ring structure of bump pads of each ring into

linearly ordered string structure => IO_connected_pad & Pad_string

Step 2 : Longest Common Subsequence

- Compute the LCS sequence using dynamic programming => LCS_string

Step 3 : Maximum Planar Subset of Chords

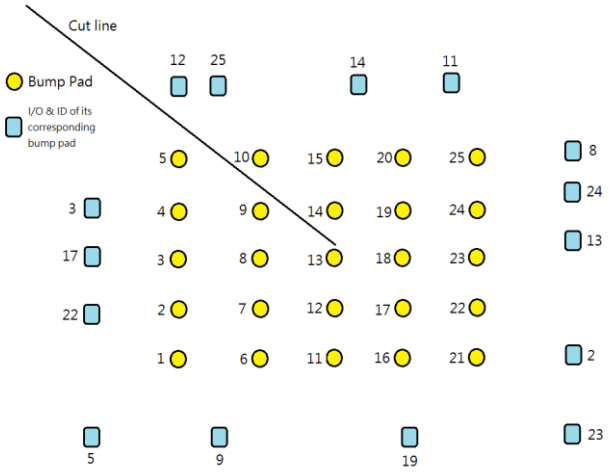
- Compute the MPSC using dynamic programming => MPSC_priority

Step 4 : Global routing

- Using the information in the previous three steps, we can derive the bump pad sequence of escape routing on each ring => Result_string

Originally I think I can use A* algorithm based on the information provided by Result_string and MPSC_priority to solve this problem. However, I did not realize that the routing order of nets within MPSC_priority can significantly affect the performance of overall algorithm, since improper routing order can result in unnecessary detours when routing nets in MPSC.

The figure below shows the test case I generate (visualization of test_rdl.txt) :




## C. *Data Structure*

- MIS & MPSC_table (MPSC)

  I defined a class MIS (maximum independent set) to store the information of each element in MPSC_table (a two–dimension array), including pointers for back trace, the cardinality of set and the integer pair corresponding to the two endpoints of the chords.

  MPSC_table[i][j] represents the MIS corresponding to the arc cut by the chord whose endpoints are node i and node j.

- Length & Prev (LCS)

  For two strings s1 and s2, Length is a two–dimension array and Length[i][j] represents the length of LCS of s1[1..i] and s2[1..j]. Prev records the information for back trace.

## *D. Result*

My RDL router cannot give the actual wire path of each net, but it can give the five terms I defined in part B.

- g++ rdl.cpp -o rdl
- ./rdl test_rdl.txt rdl_out.txt

The result will be in the file ‘rdl_out.txt’.