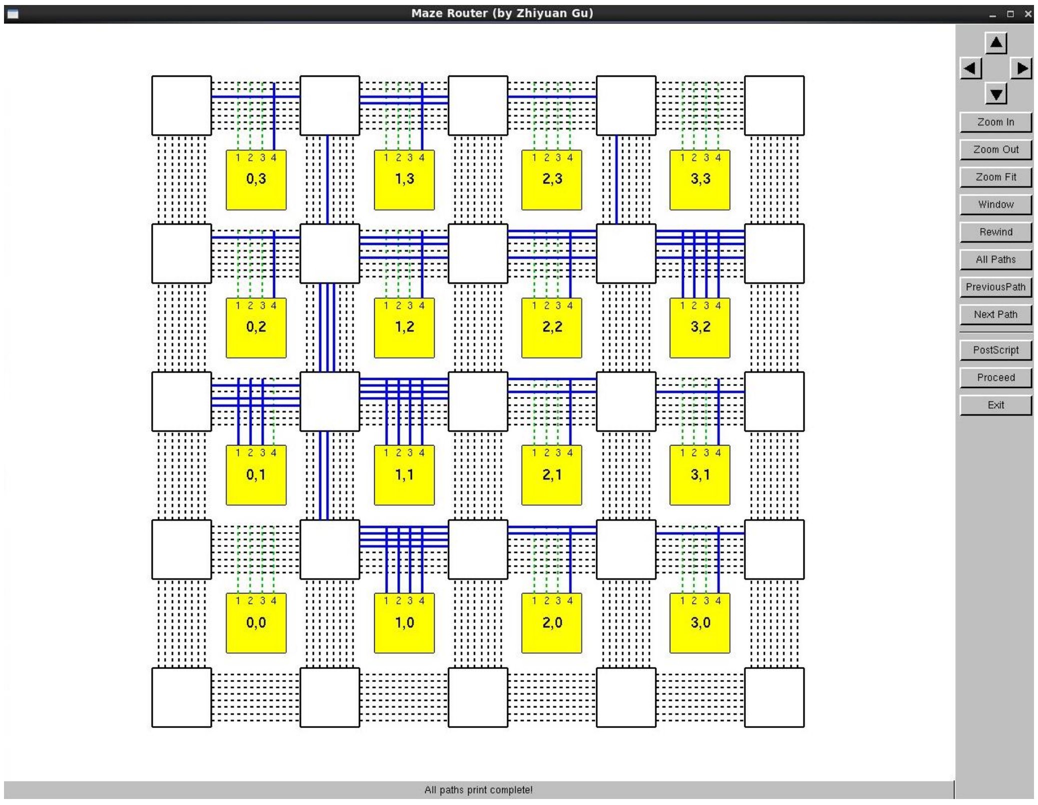
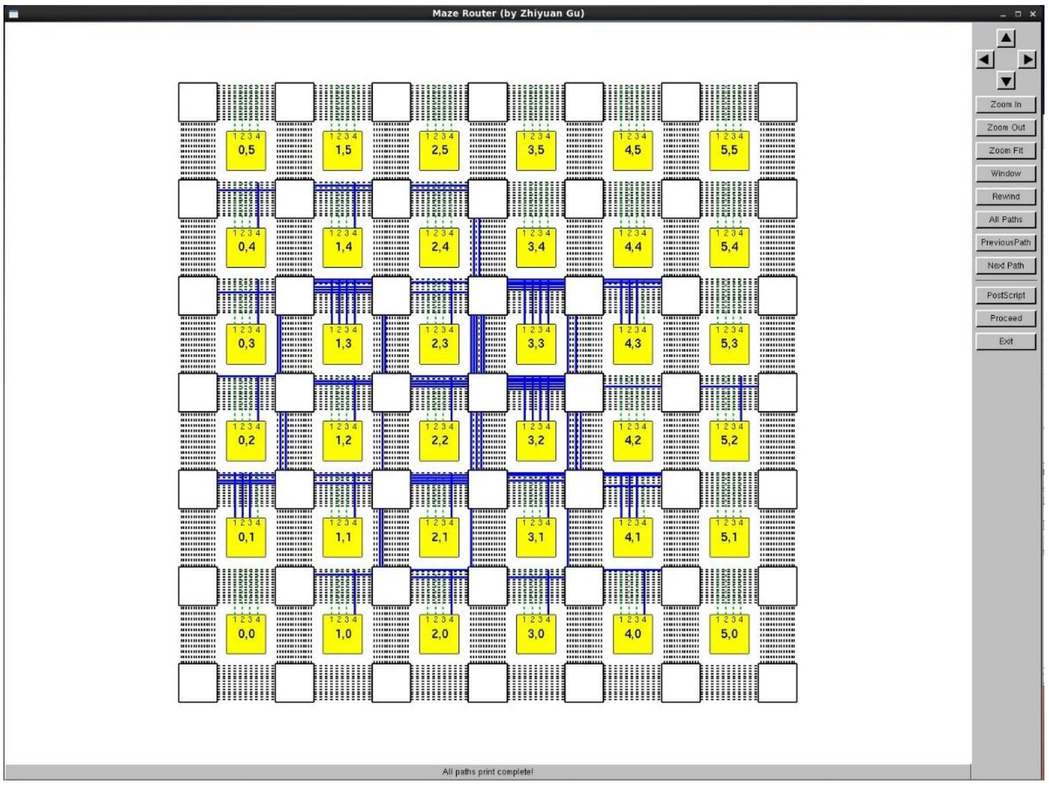
**ECE1387 assignment1 Report** Zhiyuan Gu (1004920400)

1. Paper plot

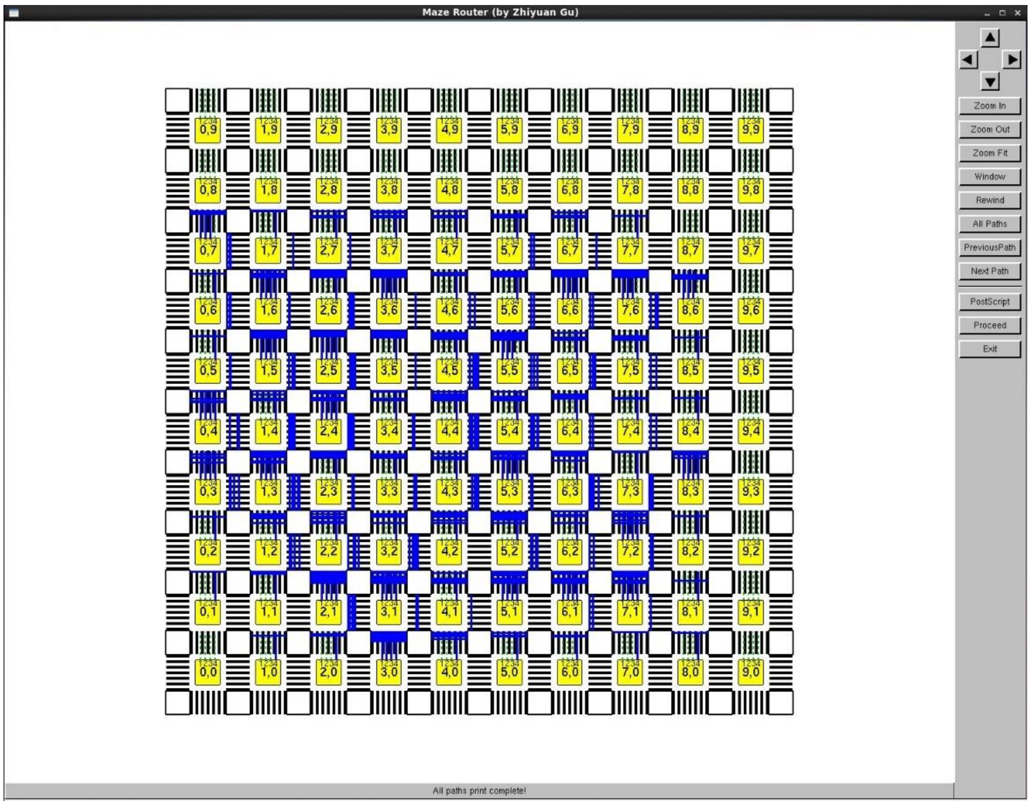
1) Test case 1



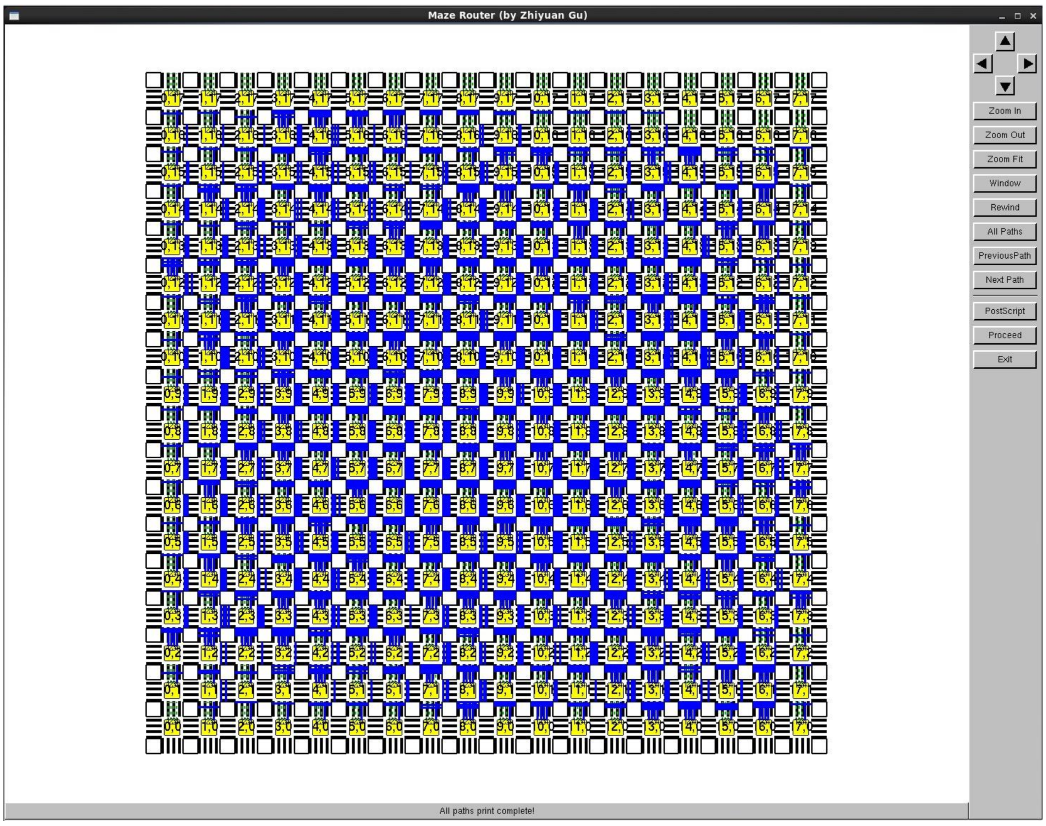
2) Test case 2



3) Test case 3



4) Test case 4



2. Total number of routing segments used in section 1

| **test case** | **cct1** | **cct2** | **cct3** | **cct4** |
|---|---|---|---|---|
| **logic grid size** | 4 | 6 | 10 | 18 |
| **channel width** | 8 | 14 | 20 | 36 |
| **number of paths to route** | 12 | 18 | 66 | 291 |
| **number of segments used** | 39 | 82 | 425 | 3629 |

3. Smallest channel width and number of segments when Wh = Wv

| **test case** | **cct1** | **cct2** | **cct3** | **cct4** |
|---|---|---|---|---|
| **channel width** | 4 | 4 | 6 | 12 |
| **number of segments used** | 39 | 97 | 485 | 4069 |

4. Smallest channel width and number of segments when allowing Wh ≠ Wv (optimize sum)

| **testcase** | **cct1** | **cct2** | **cct3** | **cct4** |
|---|---|---|---|---|
| **channel width (Wv, Wh)** | 1, 4 | 2, 4 | 3, 7 | 10, 14 |
| **number of segments used** | 40 | 101 | 484 | 3933 |

5. Optimizations applied in section 3 & 4

   1) Reusing tracks

      Reusing tracks that was marked unavailable by previous paths with the same source pin can make the most of routing resources. It is done by also checking the source of unavailable tracks when trying to find the load pin and tracing back. By allowing track reusing, it is possible to achieve less latency for the current path, create less blockages for the following paths and increase routing possibility with smaller channel width. Nevertheless, reusing was not detected in the four given test cases. It seems all the paths have different source pins.

   2) Shuffling

      Maze Routing favors the paths routed in the beginning by introducing blockages for the following paths. It is very possible that a problem path (the path that is highly likely to compete with other paths for same routing resources a lot) could appear late in the sequence of routing. With all the limitations set by previous paths, the problem path is the first to fail routing. The idea of shuffling is simple by moving the problem path to the front of the routing sequence and then route all paths with the new sequence again. If the problem path is routed without introducing new problem paths, then problem solved. Otherwise, another shuffling needs to be done to try a different sequence. The point here is to give priority to the paths that are prone to fail without it.

      Experiments has shown a quick drop in channel width requirement by applying just a few cycles of shuffling. However, large circuits with lots of paths require so many times of shuffling to get close to the optimal sequence and smallest channel width requirement. It is time consuming since the routing process for all paths needs to be repeated after shuffling. And for a strict routing grid (small channel width), shuffling several times might just introduce new problem paths every time without solving the problem.

3) Channel width loop optimization

To find the smallest routing channel width that enables successful routing of all paths, the program needs to loop through different channel widths. A naïve solution would be approaching from above which is looping from known routable larger channel widths (results from section 1) while gradually reducing width and trying to find a possible routing solution.

This solution turns out to be extremely time consuming in that Maze Routing slows down with larger channel widths. A substitution solution would be looping from small channel widths and increase widths if routing fails. For section 4, outer loop incrementation happens on the sum of two widths. The inner cycle increases one of them and decreases the other. With this change in looping sequence, runtime to find the smallest channel widths drops a lot, especially for larger circuits like test case 4.

In section 4, sum of the two widths is selected as the minimization target. One of the reasons is that this is the easiest one to apply the channel width loop optimization. Without this optimization, it could take a long time to find a good enough result. This is especially the case for runs that enable large amounts of shuffling.

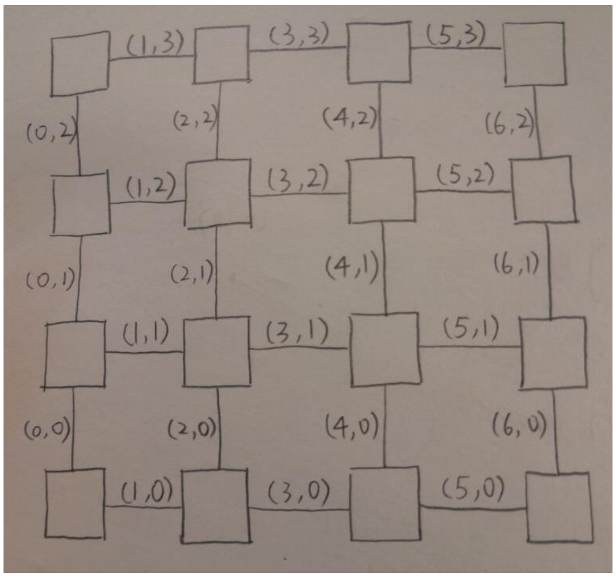
4) Manhattan distance ordering

Ordering is a big problem of Maze Routing and it is hard to determine which paths should be given the priority while leaving other paths less opportunities. Shuffling is passive solution by moving the problem path to the front when encountering a failure. It is time consuming since a new iteration of routing has to be done for all paths after shuffling. Also, it does not exploit the intrinsic priority need of paths. Solving problem of this path might create problem of another. As a result, thinking of an active reordering solution according to the requirements of paths and before routing actually happens is crucial to improve Maze Routing.

Manhattan distance ordering is a heuristic method that tries to determine the right order or sequence of all the paths to be routed by comparing the Manhattan distance between source and load pin of them. The original idea was that problem paths that appear in shuffling tend to be longer paths and they have a higher possibility to cause more congestion and fail routing. It was tried to sort all the paths according to the Manhattan distance and give longer paths more priority. However, the result turned out that this ordering slowed down the routing attempt to find the smallest channel width. One possible explanation is that satisfying demanding paths first could make it harder to meet the requirements of less-demanding paths. That is to say, the longer paths routed in the beginning could create more blockages for the following paths.

Then the opposite was tried and the results looked great. It seemed that early paths creating less blockages is critical to successful routing. To achieve the same channel widths (Wv=12, Wh=14) for test case 4 in section 4, routing without special ordering required 50 times of shuffling and 1 hour to finish routing while routing with Manhattan distance ascending ordering required shuffling only twice and could stop in 2 minutes. It seems this ordering is helping the router to quickly find a solution in grid with smaller channel widths than the original ordering could easily achieve. However, to achieve even better result that is close to optimum, the Manhattan distance ordering seems less powerful and router requires almost the same amount of time and effort as the one with no ordering to find it. This is reasonable because Manhattan distance ordering is just a heuristic representation of the priority needs of all the paths. It wins in that it does not require actual routing to determine stuff like congestion or criticality and saves runtime. But this also limits that only the process to find a suboptimal solution can be accelerated.

6. Description of software flow and design choices
   1) Class and data structure
      Every class has a separate header file and source file with the same name.
      a) *track* (class): A track is routing track segment which represents the physical tracks. Tracks appear in all the following structures so it is made into a class. It consists of its position information like position in the grid and routing information like label, availability and source pin information for track reusing.
      b) *RoutePath* (class): It represents the task to route one path and includes source and load pin information. It also includes the connection result called *usedTracks* (data structure) which is a 1D vector of tracks. These tracks form the path between source and load pin. In order to support Manhattan distance ordering mentioned above, the class has a variable as the priority given to route the path.
      c)*Routing* (class): It represents the whole routing task including steps a) to c) in the major routine section on the next page. It also includes *allPath* (data structure) which is the 1D vector of all the paths to be routed and *RoutingGrid* (data structure) which is the grid made up of tracks where the router performs Maze Routing. Logic blocks and switch blocks are not included because the grid only shows relative position of all the tracks. The grid is implemented as a 3D vector of tracks. The first two dimensions are x and y of the 2D chip plane which is shown in the picture below. By removing logic



blocks and switch blocks, only half of the original memory is needed to create the grid. The third dimension is channel index which distinguish tracks in the same channel.
      d) *q* (data structure): Expansion list implemented as FIFO queue is key to the Maze Routing algorithm which gradually increase the distance of how far the source pin can reach.

2) Major routine

The Maze Routing problem in assignment 1 can be divided into the following steps.

a) Read routing grid and paths information from test case file.
b) Create data structure by forming routing grid and routing paths.
c) Start Maze Routing and store result as tracks in routing paths.
d) Print out stored result with easygl graphics tool.
e) Loop from a) to c) with different channel widths

Step a) to step c) are the main methods to perform Maze Routing as included in *Routing* (class). Details of step c) will be introduced as follows.

Step c) can be roughly divided into subroutines as shown below:

i) label 0s and Ts which represent source tracks and load tracks
ii) pop one element from expansion list at a time
iii) find the reachable tracks from the popped one with the limit of going through only one switch block
iv) examine, label, push the tracks found in iii) and go back to ii) if not reached target
v) reached target and start tracing back using the same technique mentioned in iii)
vi) found path and store result in *usedTracks* (data structure)

Both i) and iii) are made into methods for simplicity and readability. Track reusing is implemented by taking care of unavailable same-source tracks in i) and iv). For iii), extra care was taken to stay within the grid and support different horizontal and vertical channel widths.

Step d) deals with graphics tool after Maze Routing is done so the related functions are directly called by the main function. The graphics functions are put in *draw.h* and *draw.cpp* (file) to make use of the graphic tool. A graphics window can be used to show the grid and routing result of all the paths. Several new buttons are created for easy and understandable visualization.

Step e) loops the routing process to find the smallest channel width required for successful routing.

3) Design choices

Some of the design choices have been discussed above, here are some other choices made:

a) At first, looping through different sequences of routing paths allocates a new object and deletes it every time. All history problem paths will be stored in order to shuffle correctly before next routing iteration. However, this is a waste in that previous object can be reused and previous sequence of paths can be propagated. In the new mothed, tracks in the grid and results in the paths are reset before every new iteration to avoid collision.
b) One of the possible ways to allow track reusing is to label 0 on all the tracks used by previous path with the same source pin. It is based on the idea to start routing from the previous path. However, delay caused by switch blocks and tracks seems to be overlooked by this method. It is possible that the second path from the same source pin could find a shorter path with less delay towards load pin which would not be the case if the second path starts calculating delay from half way of the previous path. In the

current version of code, used tracks with the same source are labeled with the right number indicating number of switch blocks the signal has passed rather than 0.