# Project 4: Steven Chen Portion

# Kruskal’s Generation and Random Mouse Solving Algorithms

# Design Document

## Part 1: Maze Generator

I was responsible for implementing Kruskal’s Algorithm. In reference [1] the author highlights the approach and gives an example of using Kruskal’s algorithm to construct a maze.

The basis of the algorithm is as follows:

1. Throw all of the edges in the graph into a big burlap sack. (Or, you know, a set or something.)
2. Pull out the edge with the lowest weight. If the edge connects two disjoint trees, join the trees. Otherwise, throw that edge away.
3. Repeat until there are no more edges left.[1]

## Kruskal’s Maze Generator Design and Implementation

In reference to [1] I assume all edges have the same weight and use random edge selection to select an edge. The program structure is as follows:

1. The maze is a 2D array where MAZEK[Rows][Cols]
2. Each cell in the maze is assigned an ID. Additionally all edge information for MAZEK is stored
3. Initially all edges are up (blocking the connectivity between two cells”
4. Assign each cell to a set using the Java Data Structure “Set”. All cells are also isolated initially. The Java Set data type has “add”, “clear”, “contains”, and “union” operations. These can be used to handle the set join operations from two disjointed sets. These functions are useful in implementing Kruskal’s Algorithm
5. Randomly select an edge with the “UP” status
6. Try to merge two sets connected by this edge
7. Merge two sets and update each set’s status and change the edge status to “DOWN”
8. Repeat steps 5-7 until there is only 1 set left. This end condition means that all cells have been visited and that the maze has been successfully constructed
9. Save the maze information for the solver and handle the visualization. The visualization is controlled by the delay timer, so that incremental generation can be displayed.
10. The main implementation of the code is listed below:

```
int numUnvisitedNode =numNode;
while ((numSetUsed > 1)) {
    // randomly pick an edge
    try {
        Thread.sleep(40);
    } catch (Exception e) {
        // Do nothing with errors
    }
    findEdge = 0;
    eID = -1;
    int numCheck=0;
    while(findEdge == 0) {
        eID = rand.nextInt(numEdge);// randomly pick an edge
        findEdge = checkEdge(eID);  // check if this edge is UP so
                                    // so we can put down this edge and
join two sets
    }
    eList[eID].status = EDGEDOWN;
    if(eID != -1) {
        int before =  numSetUsed;
     // apply Set’s operation to join sets if we can take down the
edge/Wall
        rts = updateSet(numSetUsed, numSet, uVisitedNode, eID);

        numSetUsed = rts[0];
        uVisitedNode = rts[1];
        if(before > numSetUsed) {
            drawFlag = 1;
        }
    }
    else {
        System.out.println("findEdge failed. Check all edge - BUG
found");
        break;
    }
}
```

Part 2: Maze Solver

I was responsible for the random mouse solving algorithm. This portion uses the data from the maze obtained in Part 1.

1. Assume the entry is node (0,0) (entryNode) and the exit is node (n-1, n-1) (exitNode) if the maze is nxn 2d Maze.
2. Call solveMaze(entryNode, exitNode)
3. Find the available neighbors with no blocked edge (UP) for each visiting node
4. Randomly pick an available neighbor cell and continue the mouse moving process recursively.
    a. There are usually four neighbors for each cell. Some cells only have one, two or three accessible neighbors because there are walls with a edge UP status that block accessibility between two cells
5. Check if the node is the exit node, return(success)
6. Repeat step 3-5 until the exit-cell is reached
7. Display the mouse moving approach progressively and animate the moving process with a different color update on cell data

Part 3: Thread Design and Implementation

A thread (taskThread) is created to manage and control Part 1 and Part 2. A timer (timer) is used to control the animation display for the Maze Generation and Solver activities. When cell or edge information is changed, the taskThread sets the drawFlag to 1. The timer will then detect the drawFlag is set to 1 and displays the Maze based on the new data, and then resets the drawFlag to 0. This is done so that it will not refresh the maze display until the drawFlag is set to 1 later on. The coordination between taskThread and timer provides a more accurate animation speed and updates the new change in real time.

Part 4: Class Design and Implementation

Point Class- This class is used to keep the cell's top left corner coordinate

Edge Class- Keeps the edge information between tow cells. It also provides the functionality for edge selection, merging, and status.

Node Class- This class is used to keep each cell's information. Supports solver() to keep cell status and related operations.

Reference:

[1] Maze Generation Algorithm - http://weblog.jamisbuck.org/2011/1/3/maze-generation-kruskal-s-algorithm

[2] Maze Solving Algorithm - https://www.robomind.net/downloads/Maze%20solving.pdf

[3] Java Set data Type - https://docs.oracle.com/javase/8/docs/api/java/util/Set.html