# Maze Generator And Solver

Jay Agrawal (181IT219)
Ravi Prakash (181IT137)
S Shushal (181IT239)
Raju Kumar(181IT236)

# INTRODUCTION

- A maze is a path or collection of paths, typically from an entrance to a goal through which the solver must find a route.
- The maze generations algorithm used in this project are based on the graph-based method.
- We wish to create a perfect maze which contains a single path solution between two different locations or nodes.

# Phases

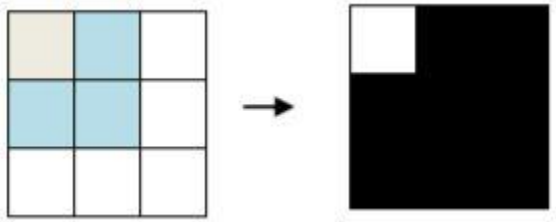
- Phase I:Maze Generation
  - Prims Algorithm
  - Kruskal Algorithm
- Phase II: Maze Solving
  - DFS
  - BFS

# Maze Generation

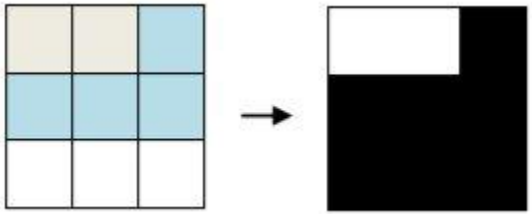
- Maze generation involves designing the layout of passages and walls within a maze
- In the tree representations of the mazes, the nodes are representing the cells and the edges are representing the walls between the cells. Since there are no “weights” in the tree, some adaptations are conducted on the original algorithms to solve the faced problems.

# A. Prim’s Algorithm Method

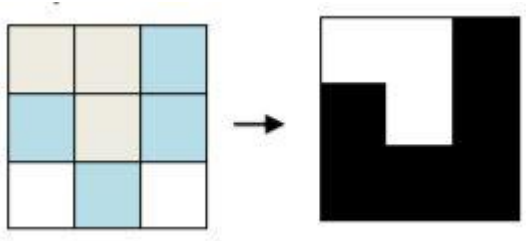
- In the 3x3 Matrix,select a cell at random and add it to the maze. Mark the added cell in the matrix.



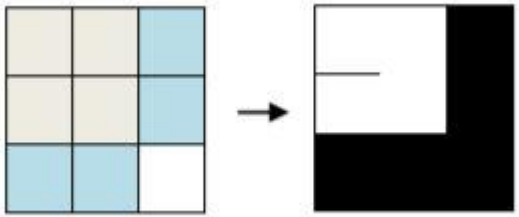
- Now, select one of the three cells (the blue-shaded cells) that are adjacent with the first cell that has been added. Add the cell to the maze.

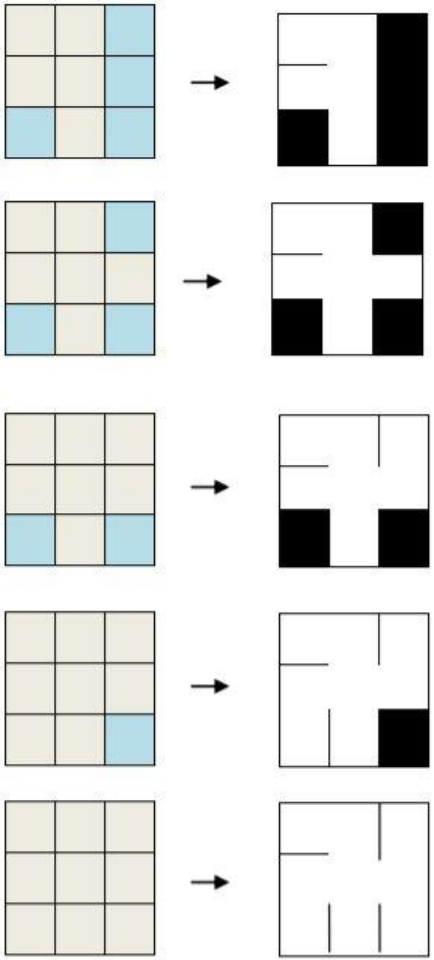


- Again, select one of the cells (the blue-shaded cells) that are adjacent with the cells that have been added previously. Add the selected cell to the maze.



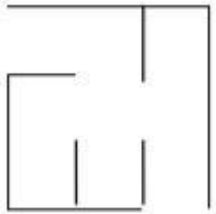
- Now, if we select a cell that is adjacent to more than one previously added cells, choose only one cell among them as the “neighbor” of the newly selected cell. Then, add the selected cell to the maze.





Repeat these processes until the maze is full.

At this point, the algorithm terminates because there are no more cells that can be selected from the matrix. Now, add “a little touch” to complete the maze, that is, add two “holes” on the outer wall to mark the start and finish point of the maze.

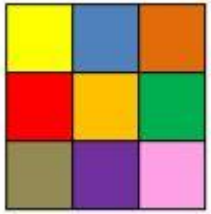


Final result of the maze.

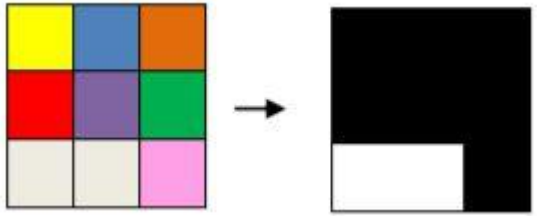
Here it is, a perfect maze generated by Prim’s algorithm is finished.

# B. Kruskal's Algorithm Method

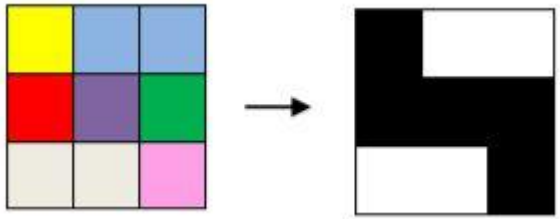
- The first step is creating a 3x3 cells matrix with a forest of 9 disjoint trees (in this case, 9 nodes). Note that different colors represent different trees or sets of cells.



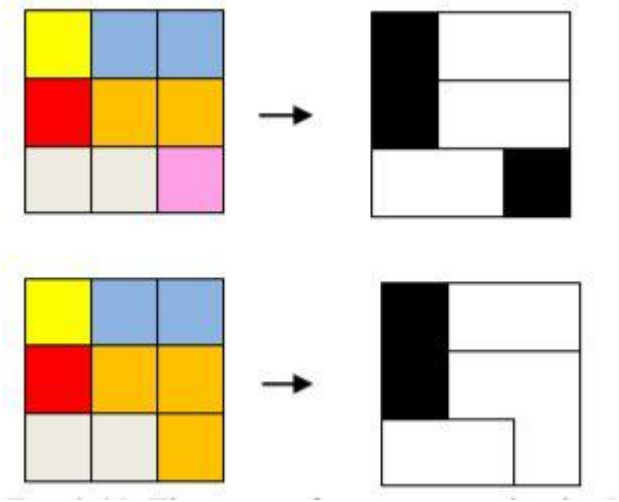
- Next, add a random edge from matrix to the maze. In this case, the added edge is the edge between cell (2,2) and (2,1).



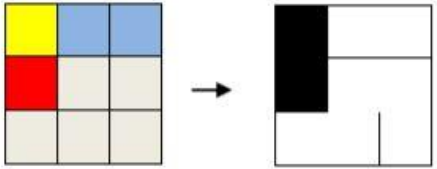
- Note that when an edge is added, two cells that are connected to it are unified into a single tree. Now, let’s add another edge to the maze.



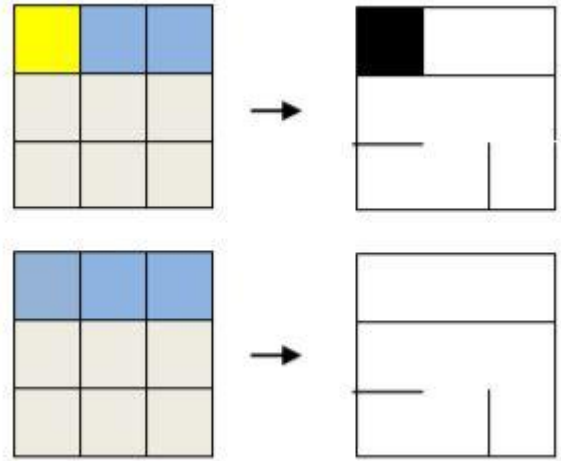
- Keep adding more edges until there is only one tree remaining in the matrix.



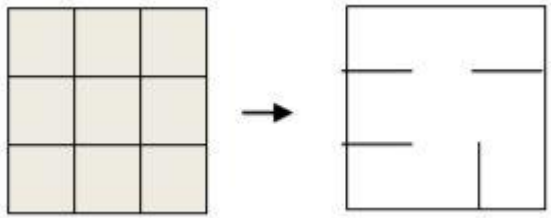
- Now add the edge between (2,2) and (3,2) so that the orange and grey tree unified into one tree.



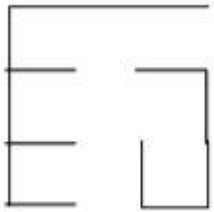
- Again, add more edges.



- At this point, the process is not yet stopped because there are still two trees left behind in the matrix. The adding of the edge between (1,2) and (2,2) unify the trees and hence, marks the final step of the algorithm.



- Finally, add random start and finish point to complete the maze.



Final result of the maze

# Maze Solving Algorithm

- **Depth First Search (DFS)**
- **Breadth First Search (BFS)**

# DFS

- The defining characteristic of this search is that, whenever DFS visits a maze cell *c*, it next searches the sub-maze whose origin is *c* before searching any other part of the maze.
- This is accomplished by using a Stack which is implemented using queues to store the nodes.
- The end result is that DFS will follow some path through the maze as far as it will go, until a dead end or previously visited location is found.
- When this occurs, the search backtracks to try another path, until it finds an exit.

# BFS

- The defining characteristic of this search is that, whenever BFS examines a maze cell *c*, it adds the neighbors of *c* to a set of cells which it will to examine later.
- In contrast to DFS, these cells are removed from this set in the order in which they were visited; that is, BFS maintains a *queue* of cells which have been visited but not yet examined (an examination of a cell *c* consists of visiting all of its neighbors).
- BFS is implemented using a Queue for the set.
- The end result is that BFS will visit all the cells in order of their distance from the entrance. First, it visits all locations one step away, then it visits all locations that are two steps away, and so on, until an exit is found. Because of this, BFS has the nice property that it will naturally discover the shortest route through the maze.

# ANALYSIS

**MAZE GENERATOR:**

| Algorithm | Dead End | Memory | Time | Solution |
|---|---|---|---|---|
| Kruskal | 30% | $N^2$ | 21 | 4.1% |
| Prim | 36% | $N^2$ | 43 | 2.3% |

- The dead end statistic measures the approximate percentage of dead ends cells upon of the total cells.
- The memory statistic represents how much extra memory is required to implement the maze with NxN cells.
- The time measures the relative time needed to create one maze with the lower number being faster (the fastest is speed 10).
- The solution statistic represents the percentage of cells in the maze that the solution path passes through.

- Time Complexity: Kruskal’s is quite faster.
  - This happens because Kruskal’s maze generate less dead-ends than Prim’s.

- Complexity of Solution Path:
  - Kruskal’s maze tends to have longer and more “windy” solution path than the Prim’s.
  - This stat is related to the fact that Prim’s generated maze tends to have more dead-ends and thus, have less cells in the solution path than the Kruskal’s maze.

**MAZE SOLVER:**

- Space Complexity :
  - More Memory is required for BFS than DFS
- Time Complexity:
  - DFS is faster than BFS
  - O(R+C) for BOTH DFS and BFS algorithm
    - Where R is number of rows.
    - Where C is number of columns.

# RESULTS

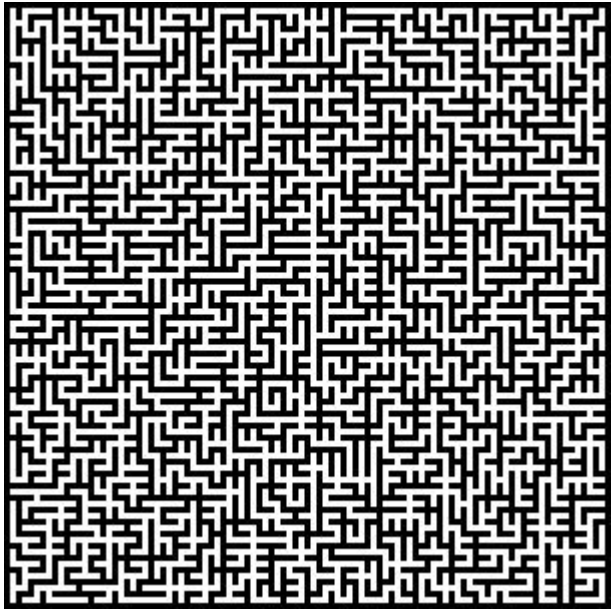


Figure: maze created using kruskal

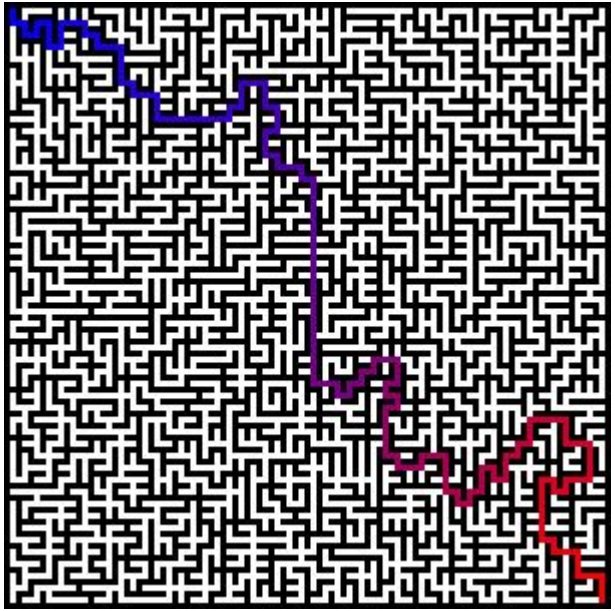


Figure : Maze solved using DFS

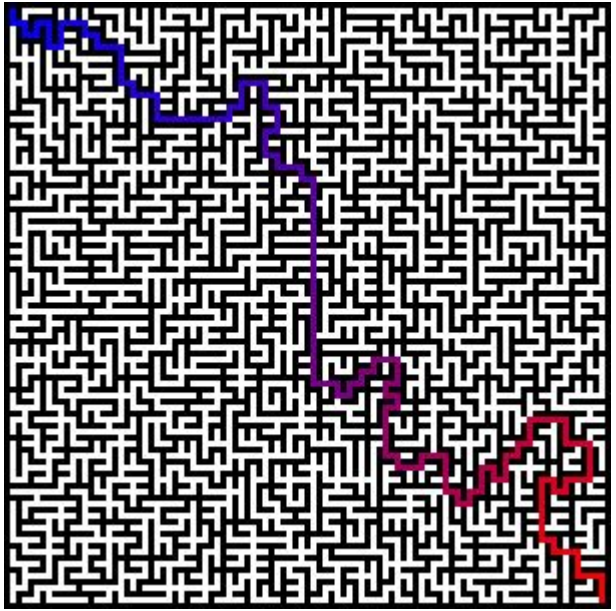


Figure : Maze solved using BFS

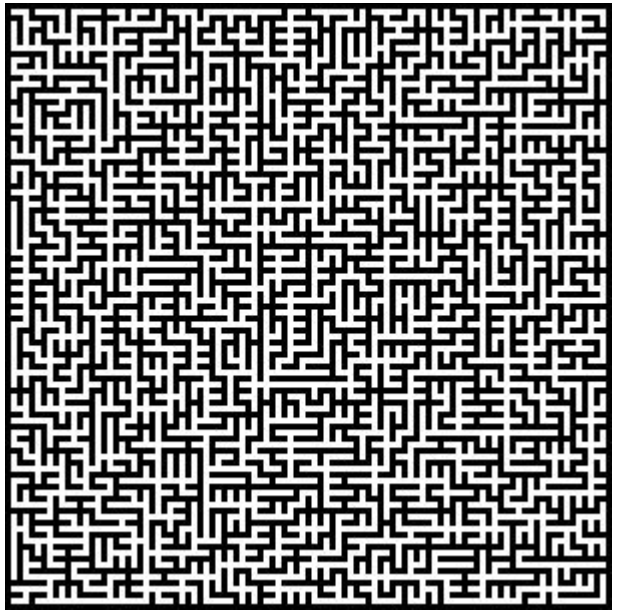


Figure: maze created using prim

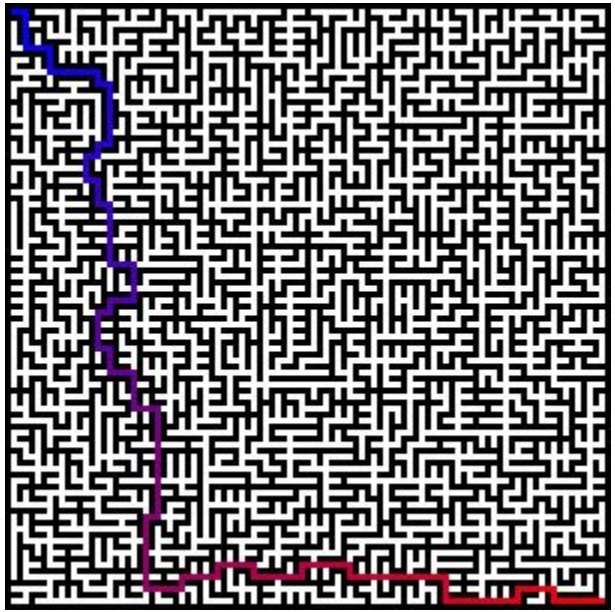


Figure : maze solved using DFS

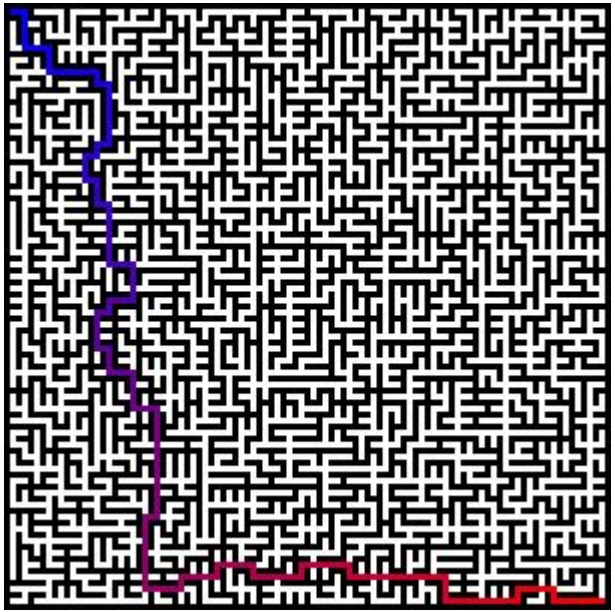


Figure : maze solved using BFS

# CONCLUSION

- Prim’s algorithm generates the maze by placing cells one by one from the cells matrix to the maze,
- Kruskal’s algorithm proceeds by placing the edges one by one.
- If a complex and hard-to-solve maze is wanted, Prim’s algorithm is the choice. But, if the processing time is more important, Kruskal’s algorithm is better.
- Prim’s Maze tend to have higher complexity than Kruskal’s because it has more dead ends and less solution percentage.

# CONCLUSION

- Coming to maze solving part, The end result of DFS will follow some path through the maze as far as it will go, until a dead end or previously visited location is found. When this occurs, the search backtracks to try another path, until it finds an exit.
- The end result of BFS will visit all the cells in order of their distance from the entrance. First, it visits all locations one step away, then it visits all locations that are two steps away, and so on, until an exit is found. Because of this, BFS has the nice property that it will naturally discover the shortest route through the maze.

# REFERENCES


- https://github.com/jsmolka/pyprocessing for visuals
- https://www.researchgate.net/publication/305652371_Implementation_of_Prim's_and_Kruskal's_Algorithms'_on_Maze_Generation