Implementation and Analysis of Maze Generating Algorithms

Submitted by ***Shaikh Ubaid (180001050)***

Computer Science and Engineering

2nd Year

03-06-2020

Under the Guidance of

Dr. Kapil Ahuja

Department of Computer Science and Engineering
Indian Institute of Technology Indore
Spring 2020

ACKNOWLEDGEMENT

I am highly indebted to Dr. Kapil Ahuja, Associate Professor, IIT Indore for his guidance and constant supervision as well as for providing an opportunity to work on Algorithm Analysis in real life.

I am thankful to the TAs, Mr. Rohit Aggrawal and Mr. Siddharth Gupta, who guided me and supported me from the initial to the final level.

I have gained a lot of knowledge through this project, and I have had an enriching experience in this research project.

OVERVIEW & PURPOSE

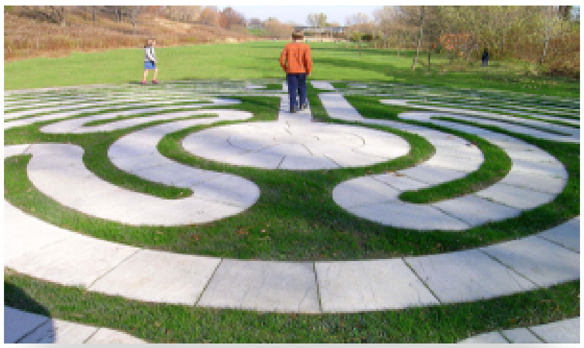
- Goal- The main goal of this project is to rank different maze generating algorithms according to the difficulty of the generated mazes.
- Following our main goal, we implement and analyse four maze generating algorithms.
- For the purpose of evaluating and ranking maze generating algorithms, we devise four agents which solve mazes and report the results.
- To assess the level of difficulty of a maze we inspect several features such as the number of visited intersections, dead ends, and overall steps of the agents.
- According to agents performances, we rank maze generating algorithms.
- The best performing algorithms are derived from algorithms for finding uniform spanning trees in graphs.

CONTENT

1. INTRODUCTION

1.1 History

Mazes are closely related to labyrinths which have been known since ancient times. Usually, they were built with naturally occurring materials. Originally they have had a spiritual connotation. Their later purpose was mainly amusement. In modern times, mazes became intriguing for scientists, especially for mathematicians.



1.2 Uses

Mazes are used in various fields besides entertainment purposes, mazes are also used in psychology studies of human and animal behaviour to determine space awareness and also intelligence. Among others, mazes can be used in physics, for example in the study of crystal structures.

1.3 Project Flow

We describe and analyse four maze generating algorithms. Our algorithms are originally used in graph theory. By using a specialisation method we implemented them to generate mazes.

Perfect maze - A so-called 'perfect' maze has every path connected to every other path, so there are no unreachable areas. Also, there are no path loops or isolated walls. There is always one unique path between any two points in the maze.

We generate mazes which are represented as trees only. Our focus is on 2D and planar mazes that do not contain overpasses

We devise four types of agents that walk across the mazes to help us analyse the difficulty of each maze type. They tell us the number of steps they take from the beginning to the end as well as the number of visited intersections and dead ends.

Finally, we analyse the relation between the properties of mazes and attributes that agents provide us with. The final goal is to determine which algorithms are giving us the most difficult mazes.

In the next section, we describe four maze generating algorithms. In section 3 we establish the maze solving agents and their properties. In section 4 we analyse mazes. In section 5 we have a discussion. In section 6, we present the results.

Note: The codes for all the algorithms, agents, data structure discussed in this report are given in the Appendix at the end of the report.

2. GENERATING MAZES

Let us first explain the data structure that we are using to represent maze. We start with the square grid graph. Initially, all the edges (connections) represent walls. The algorithms convert specific walls into passages. After the algorithms do their job, the subgraph made out of passages represents a tree-like structured maze. In general, mazes can contain loops, overpasses, etc., but we focused on simple ones.

Every maze has properties which we use in the analysis, such as size, number of intersections, number of branches, average branch length and length of the dead ends (branches that do not split further).

Note- All mazes analysed are square.

2.1 Recursive Backtracking (RB)

Firstly, we implement the depth-first search (DFS) algorithm also called backtracking. The main idea of DFS is to go forward as much as possible, then backtrack to the first branch that has unvisited paths and repeats until everything is searched. We use randomised DFS to obtain random (non-trivial) mazes. Basic implementation of DFS uses recursion. However, we implemented it using an explicit stack to avoid stack overflow error caused by large mazes having long paths.

Pseudo Code/Steps:-

1. Choose a starting point in the field. (used the 1st grid in implementation)
2. Randomly choose a wall at that point and carve a passage through to the adjacent cell, but only if the adjacent cell has not been visited yet. This becomes the new current cell.
3. If all adjacent cells have been visited, back up to the last cell that has uncarved walls and repeats.
4. The algorithm ends when the process has backed all the way up to the starting point.
   As this algorithm visits every cell of the gird, its complexity turns out to be O(n^2) where n is the size of the grid.

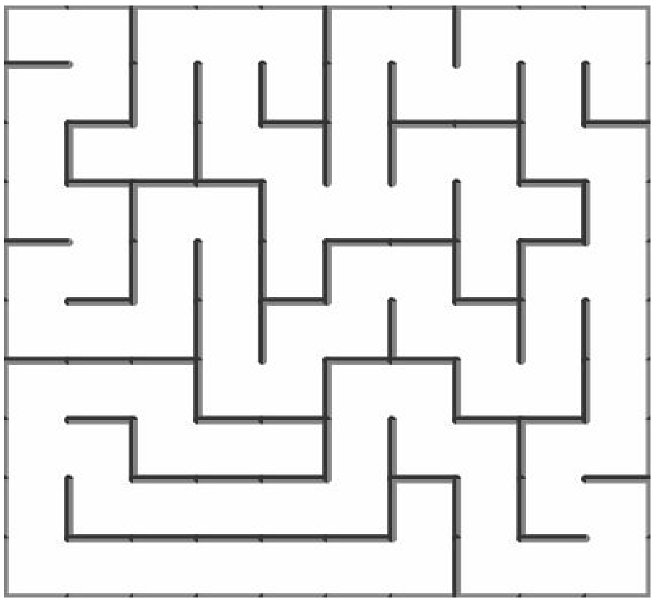


Figure 1: Maze generated by Recursive Backtracking Algorithm

2.2 Aldous-Broder Algorithm (AB)

Aldous-Broder uses a random walk until all vertices are visited. During the walk, suitable connections between the vertices are created (under certain criteria). This algorithm is originally used to find uniform spanning tree in the graph.

Pseudo Code/Steps:-

1. Choose a vertex. Any vertex.
2. Choose a connected neighbour of the vertex and travel to it. If the neighbour has not yet been visited, add the travelled edge to the spanning tree.
3. Repeat step 2 until all vertices have been visited.

As this algorithm and the next one are random walking, their worst-case complexities could be infinite. Hence we study their time comparison graphs.

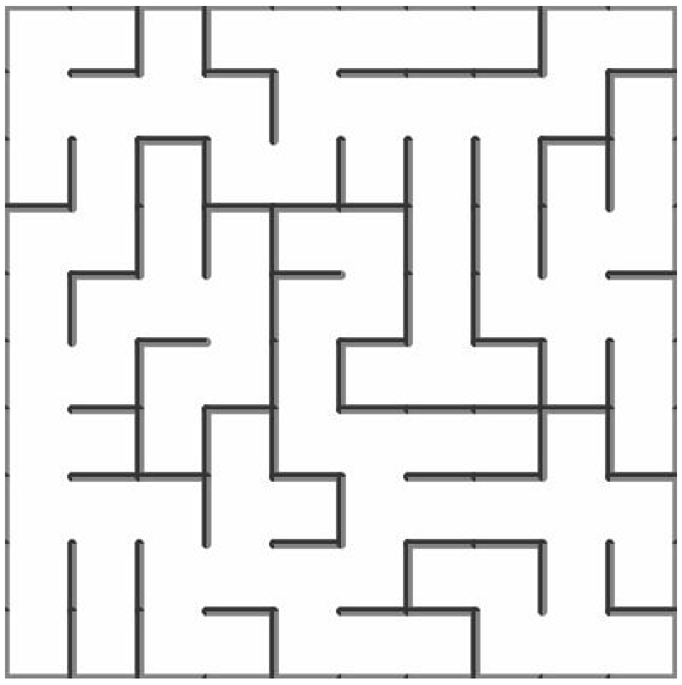


Figure 2: Maze generated by Aldous Broder Algorithm

2.3 Wilson’s Algorithm (W)

Wilson’s algorithm is originally used to find a uniform spanning tree in the graph. The algorithm is very similar to Aldous-Broder with slightly better asymptotic time-complexity in theory. In practice, Wilson turns out to be slower because it uses dictionaries unlike the Aldous-Broder, which uses only arrays.

Pseudo Code/Steps:-

1. Choose any vertex at random and add it to the UST.(Uniform Spanning Tree)
2. Select any vertex that is not already in the UST and perform a random walk until you encounter a vertex that is in the UST.
3. Add the vertices and edges touched in the random walk to the UST.
4. Repeat 2 and 3 until all vertices have been added to the UST.

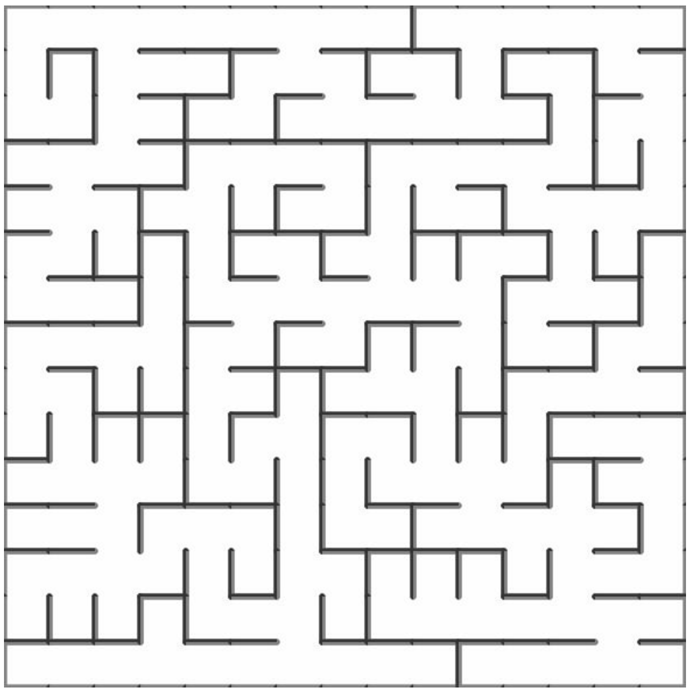


Figure 3: Maze generated by Wilson Algorithm

2.4 Hunt and Kill (HK)

The Hunt and Kill algorithm uses the idea of the recursive backtrack but it starts from a random unvisited vertex whenever it hits the dead end. It does not backtrack to the last vertex with unvisited neighbours. There is an iterative version of this, which starts from the first unvisited vertex which has a visited neighbour.

Pseudo Code/Steps:-

1. Choose a starting location.
2. Perform a random walk, carving passages to unvisited neighbours, until the current cell has no unvisited neighbours.
3. Enter "hunt" mode, where you scan the grid looking for an unvisited cell that is adjacent to a visited cell. If found, carve a passage between the two and let the formerly unvisited cell be the new starting location.
4. Repeat steps 2 and 3 until the hunt mode scans the entire grid and finds no unvisited cells.

Again this algorithm visits each cell once, hence time complexity is O(n^2)

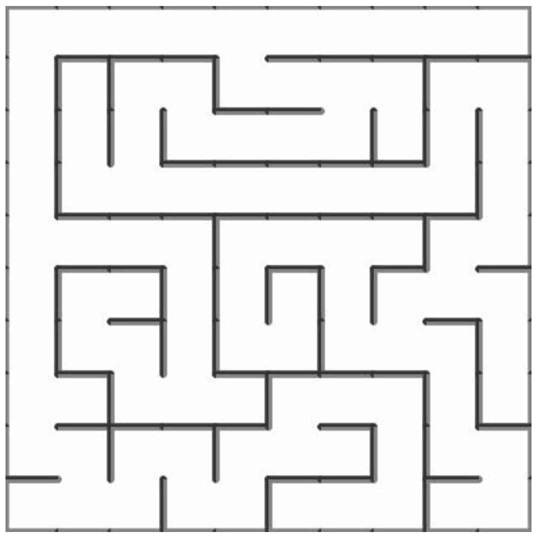


Figure 4: Maze generated by Hunt and Kill Algorithm

3. Solving Agents

Agents help us understand how difficult a particular maze is to solve. Agents produce various attributes of mazes with which we will later analyse the difficulty of the mazes.

These attributes are:

- number of steps needed from the start to the end of maze
- number of visited cells
- number of visited intersections
- number of visited dead ends

3.1 Random Walk (RW) Agent

The agent walks randomly from a vertex to its random neighbour until it gets to the end of the maze. In particular, when located in a node, an agent selects a neighbouring node uniformly at random and moves into it. It repeats this procedure until it finds the end.

3.2 Depth First Search (DFS) Agent

This agent walks as far as it can until it hits a dead end. The agent then backtracks to the first node with unvisited neighbours. It keeps repeating the walk until it comes to the end of the maze. The precedence of agent's turns at intersections are manually predefined: west, north, east, south.

3.3 Heuristic Depth First Search (HDFS) Agent

Similar to the DFS agent, but selects the preferred directions with a simple heuristic. In particular neighbours with lower Manhattan distance to the end of the maze are preferred.

3.4 Breadth-First Search (BFS) Agent

This agent uses the idea of BFS to solve the maze but instead of the queue to visit the nodes the agent first visits the nodes closest to the end of the maze. This agent resembles a human solver which can freely jump from one path to another (at least when solving printed mazes).

4. Analysis

In this section, we analyse the difficulty of the mazes constructed by the above four generating algorithms. We analyse the properties of mazes and results of solving agents to determine the difficulty of mazes. For the sake of completeness we also experimentally analyse the time performance of algorithms.

4.1 Maze Generating Time

We analyse the execution time of maze generating algorithms with respect to the maze size. We did not bother to consider the number of nodes of the maze which would be a true measure. We used only the length of the side of the square grid graph because we are interested in relations among performances of different algorithms. The result is shown in Figure 5 on the next page.

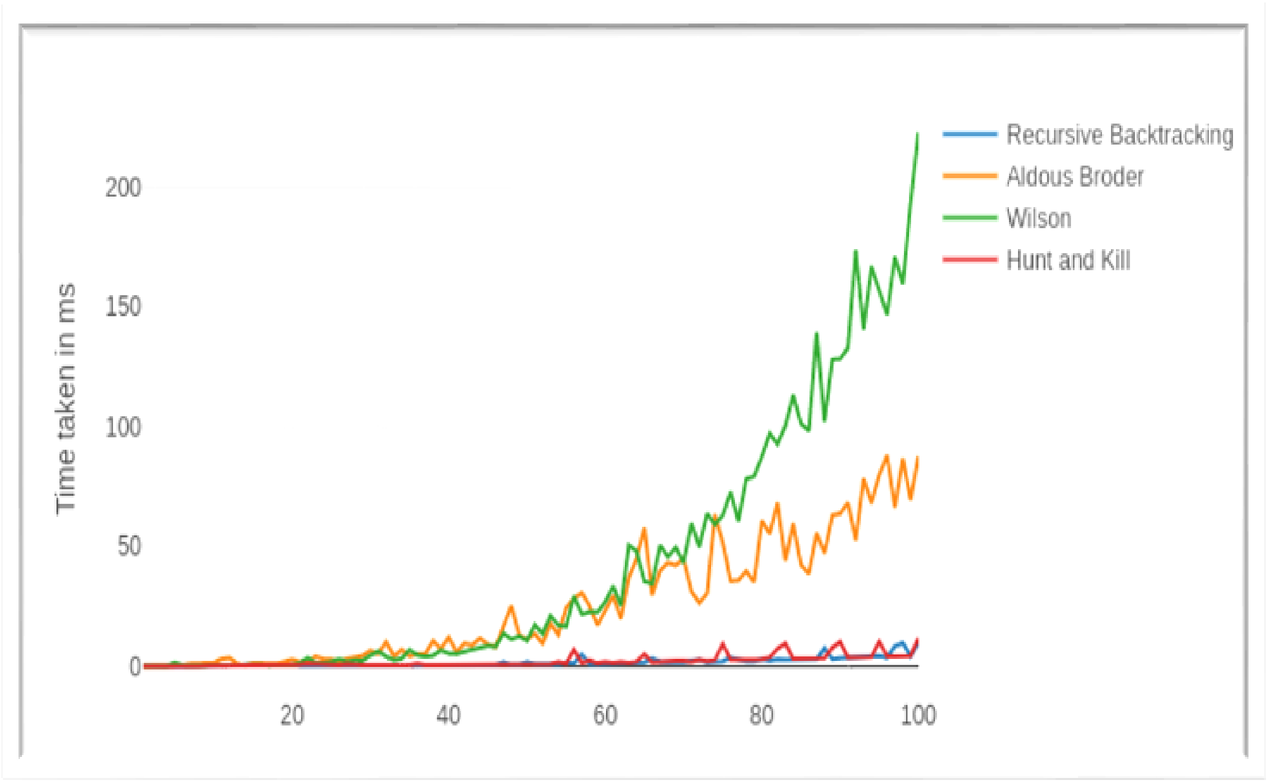


Figure 5: Running time of the maze generating algorithms with respect to the maze size.

Conclusions from the Graph:

Some algorithms stand out performance-wise. They are either exceptionally slower or faster than we would expect.

Wilson algorithm is very similar to Aldous-Broder with slightly better asymptotic time-complexity in theory. In practice, Wilson turns out to be slower because it uses dictionaries unlike the Aldous-Broder, which uses only arrays.

We have Aldous- Broder algorithm which is surprisingly fast. It is simple and uses primitives instead of higher data structures.

As both Recursive Backtracking and Hunt and Kill algorithms basically use DFS, their time complexity curves are similar.

4.2 Maze Properties

To analyse the difficulty of a maze, we consider the following properties:

- size s
- number of intersections n_i; intersections are vertices with more than two neighbours
- number of dead ends n_{de}; dead ends are vertices with only one neighbour.

The bigger n_i, the more difficult it is to solve the maze. The same goes for dead ends.

I generated 100 mazes (of size 20 × 20) of each type and calculated the number of intersections and dead ends. The average results are shown in Table 1.

	No of Intersections Generated	No of Dead Ends Generated	Score	Rank
AldousBroder	10173	11709	2	1
Wilson	10147	11663	2	1
RecursiveBacktracking	3973	4216	0.753029337	2
HuntAndKill	3728	3938	0.7027827249	3

Table 1: Average number of intersections and dead ends of the mazes.

Table 1 indicates that certain mazes generated with similar algorithms behave similarly. In particular, Aldous-Broder and Wilson have practically the same number of intersections and dead ends. They both originate from algorithms for finding uniform spanning tree.

Hunt and Kill and Recursive backtrack have notably smaller values of n_i and n_{de} than other mazes. These algorithms originate from DFS.

The larger number of intersections and dead ends means more difficult maze, one can deviate from a correct path easily.

Aldous Broder and Wilson perform best in this case.

4.3 Agent Performance

Maze solving agents give us another set of maze properties:

- number of steps s that agent needed to get from the beginning to the end
- number of visited intersections $i_{v,}$
- number of visited dead ends $de_{v.}$

4.3.0 Ranking

We ranked maze generating algorithms according to the performances of the maze solving agents. To rank maze generating algorithms we devised a simple method. For every algorithm i, we calculated score $s_{i:}$

$$s_i = \sum (A_i \div max(A))$$

where A_i represents the value of the score of an agent A for algorithm i, and max(A) represents the maximum value that the agent scored among all generating algorithms. According to the score s_i, we ranked generating algorithms. Algorithm with the biggest (best) score has rank 1, etc.

4.3.1 Number of Steps

The basic measure is the number of steps an agent makes from the start to the end. A step is defined as a transition from a node to the adjacent node.

	DFS	BFS	HeuristicsDFS	RandomWalk	Score	Rank
RecursiveBacktracking	33940	24038	16199	4819386	3.693617267	1
Wilson	29585	33568	13346	1715718	3.03706382	2
AldousBroder	29654	34062	12055	1994888	3.886127007	3
HuntAndKill	17055	34656	11427	1976270	2.617985094	4

Table 2: Average number of steps needed from the start to the end.

4.3.2 Visited Intersections

Next, we analyse how many intersections agents visit. The more intersections that an agent visits the better chance to miss the right path. Hence, the generating algorithm is more difficult.

	DFS	BFS	HeuristicsDFS	RandomWalk	Score	Rank
RecursiveBacktracking	33940	24038	16199	4819386	3.693617267	1
Wilson	29585	33568	13346	1715718	3.03706382	2
AldousBroder	29654	34062	12055	1994888	3.886127007	3
HuntAndKill	17055	34656	11427	1976270	2.617985094	4

Table 3: Average number of visited intersections of each agent.

We used the same ranking technique as in section 4.3.1. Unlike the number of steps, here Hunt and Kill performs badly. The best performing algorithms are Aldous-Broder and Wilson, whose original idea is finding uniform spanning trees in graphs.

4.3.3 Visited Dead Ends

The last property that we analyse is the number of dead ends that agents visit on average.

	DFS	BFS	HeuristicsDFS	RandomWalk	Score	Rank
AldousBroder	8187	9718	2718	294181	3.892024943	1
Wilson	8059	9557	3047	253293	3.828809001	2
RecursiveBacktracking	3188	2120	627	219894	3.367494267	3
HuntAndKill	1213	3343	855	97047	1.102655158	4

Table 4: Average number of visited dead ends of each agent.

5. Discussion

Our goal was to rank the maze generating algorithms from those that generate the most difficult mazes to those that generate the least difficult mazes.

We did that with the help of two criteria:

1. maze properties and
2. solving agents performances.

6. Results

We ranked the algorithms according to the criteria. The final ranking of difficulty level: For every measure, we ranked algorithms. Finally, we calculated the average of all the ranks which gives us the final order in Table 5.

	Steps were taken by different agents	Intersections Visited by different agents	Dead Ends Visited by different agents	TotalIntersections and Dead EndsGenerated	Average	Rank
AldousBroder	3	1	1	1	1.5	1
Wilson	2	2	2	1	1.75	2
RecursiveBacktracking	1	3	3	2	2.25	3
HuntAndKill	4	4	4	3	3.75	4

Table 5: Ranking of algorithms by the level of the difficulty of Mazes generated.

Having established the ranking of the algorithms we can now find the properties that distinguish among the various levels of difficulty of the algorithms.

The number of intersections is correlated to the difficulty of mazes. More intersections mean that the maze is more difficult and that there is more chance to miss the correct path. The type of algorithm contributes to the level of the difficulty:

• Best results are achieved by Aldous- Broder and Wilson. They originate from algorithms for finding uniform spanning trees in graphs. We speculate that agents have difficulty navigating through the maze because the paths are unbiased in any direction.

• The worst performing pair is Recursive Backtracking, and Hunt and Kill algorithms. They originate from the graph search algorithms. On the other hand, most solving agents use the same approach which enables them to solve the maze easily. Therefore the mazes are generated in a way that suits the solving agents.

7. Combining Aldous Broder and Wilson

Problem Tackled

In Wilson's Algorithm, it takes too much time initially to find paths to already visited vertices as it uses loop-erased random walk and its loops are frequently erased in the beginning as the number of visited vertices is less in the beginning and hence, finding a path to a visited vertex is difficult.

In Aldous Broder, it takes too much time in the end to visit the unvisited vertices, as it is a random walking algorithm and does not have any sense of direction/attraction towards the unvisited vertices located in corners.

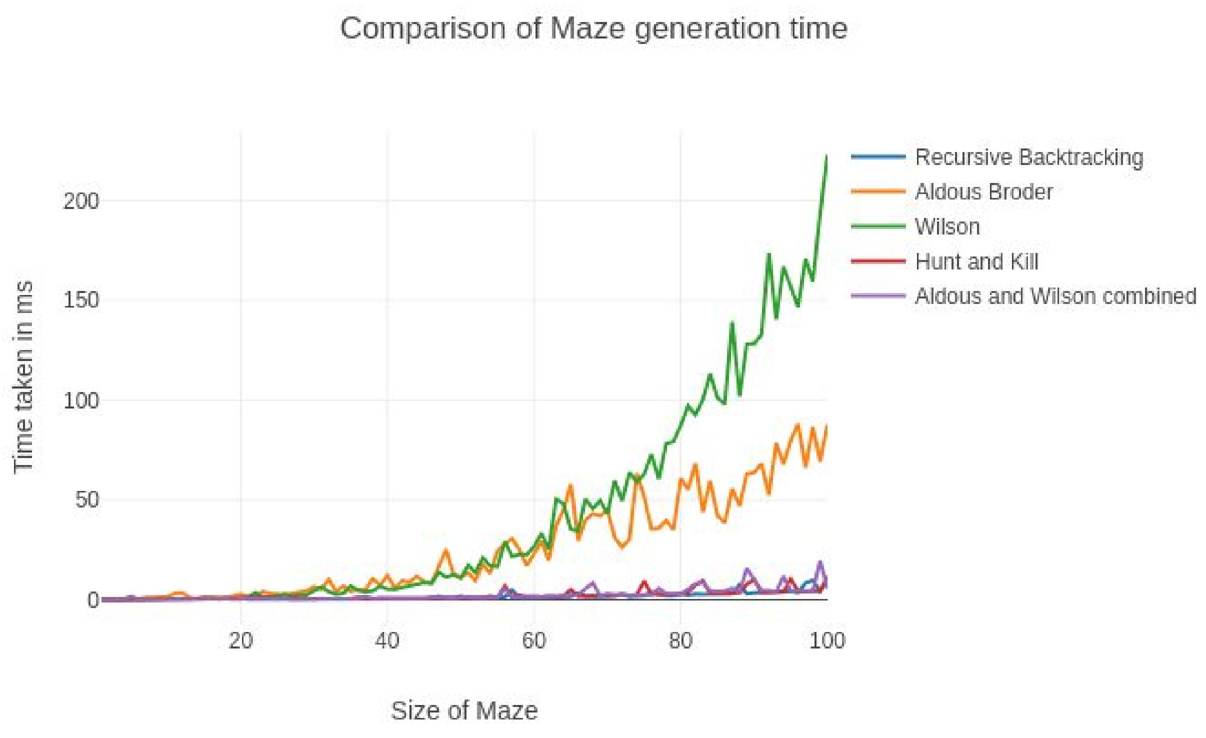
Solution proposed

The idea is to combine both Aldous-Broder and Wilson's Algorithm. Doing AB until about 30% of the field is filled, and then switching to Wilson's which empirically improves the odds quite a bit.

Why this solution is better

As we will be doing Aldous Broder initially, it won't take much time to fill about 30% of the maze (that is visiting at least 30% of the available vertices) as this algorithm is fast in the beginning. Then we switch to Wilson's Algorithm, which will also work fast as it has started from a state where the density of visited vertices is more. Aldous Broder is not good at the end and Wilson is not good at the beginning. So we use Aldous at the beginning and later Wilson so that the drawbacks of both are eliminated and the benefits of both are combined.

Graph After Combining both the Algorithms



Here, we can see clearly that the Combined Algorithm takes much less time as compared to Aldous Broder and Wilson's Algorithm.

Note: Code for Combined Algorithm is given in the Appendix.

8. Conclusion and Future

In our project, we studied and analysed two different approaches of generating mazes and were able to rank them by levels of difficulty. Nevertheless, both two considered types were somehow kindred since they are used for finding trees in graphs. In the future, it would be useful to take into consideration algorithms with a completely different approach and then compare the results.

As a continuation of our work, it would be worthwhile looking into more complex mazes such as spatial, braided, overlapping, etc.

9. References

1. Gabrovˇsek, Peter, "Analysis of Maze Generating Algorithms", The IPSI BgD Transactions on Internet Research, January 2019 Volume 15 Number 1 (ISSN 1820-4503).
2. Wikipedia contributors. "Maze generation algorithm" [Internet]. Wikipedia, The Free Encyclopedia; 2020 May 31.
3. A. Karlsson, "Evaluation of the Complexity of Procedurally Generated Maze Algorithms," Dissertation, 2018.
4. Jamis Buck, "Maze Generation: Algorithm Recap", the Buckblog 7 February 2011.
5. Ms. Shivani H. Shah, Ms. Jagruti M. Mohite, Mr. Anoop G. Musale, Mr. Jay L. Borade, "Survey Paper on Maze Generation Algorithms for Puzzle Solving Games", International Journal of Scientific & Engineering Research, Volume 8, Issue 2, February-2017 (ISSN 2229-5518).

10. Appendix

Initial Setup

```
function setup() {
  size=400;
  w=40;
  createCanvas(size, size);
  // frameRate(5);
  background(255, 204, 0);
  rows = floor(size / w);
  cols = floor(size / w);
  for(let r=0;r<rows;r++)
  {
    for(let c=0;c<cols;c++)
    {
      var cell = new Cell(r,c);
      grid.push(cell);
    }
  }
  AnyAlgorithm();     //Calling the respective Algorithms here
  console.log("AnyAlgorithm Completed");
}
```

Grid Structure

```
function Cell(i, j) {
    this.i = i;
    this.j = j;
    this.index = Index(i, j);
    this.walls = [true, true, true, true];
    this.visited = false;
    this.parent = -1;
    this.checkNeighbours = function () {

        var neighbours = [];
        var top = grid[Index(i - 1, j)];
        var right = grid[Index(i, j + 1)];
        var bottom = grid[Index(i + 1, j)];
        var left = grid[Index(i, j - 1)];
        if (top && !top.visited)
            neighbours.push(top);
        if (right && !right.visited)
            neighbours.push(right);
        if (bottom && !bottom.visited)
            neighbours.push(bottom);
        if (left && !left.visited)
            neighbours.push(left);
        return neighbours;
    }
    this.checkNeighboursUltimate = function () {
        var neighbours = [];
        var top = grid[Index(i - 1, j)];
        var right = grid[Index(i, j + 1)];
        var bottom = grid[Index(i + 1, j)];
        var left = grid[Index(i, j - 1)];
        if (top)
            neighbours.push(top);
        if (right)
            neighbours.push(right);
        if (bottom)
            neighbours.push(bottom);
        if (left)
            neighbours.push(left);
        return neighbours;
    }
    this.checkVisitedNeighbour = function () {
        var top = grid[Index(i - 1, j)];
        var right = grid[Index(i, j + 1)];
        var bottom = grid[Index(i + 1, j)];
        var left = grid[Index(i, j - 1)];
        if (top && top.visited)
            return top;
        if (right && right.visited)
            return right;
        if (bottom && bottom.visited)
            return bottom;
        if (left && left.visited)
            return left;
        return false;
    }

    this.markVisited = function () {
        this.visited = true;
    }
    this.markVisitedUltimate = function () {
        this.visited = true;
        Delete(this);
    }
    this.connect = function (u) {
        var value = this.index - u.index;
        if (value == 1) {
            this.walls[3] = false;
            u.walls[1] = false;
        }
        if (value == -1) {
            u.walls[3] = false;
```

```
                this.walls[1] = false;
            }
            if (value == cols) {
                this.walls[0] = false;
                u.walls[2] = false;
            }
            if (value == -cols) {
                u.walls[0] = false;
                this.walls[2] = false;
            }
        }
    }
```

Helping Functions

```
    function Random(array) {
        return array[Math.floor(Math.random() * array.length)]
    }


    function Index(i, j) {
        if (i < 0 || j < 0 || i >= rows || j >= cols)
            return -1;
        return i * cols + j;
    }

    function shuffleArray(array) {
        for (var i = array.length - 1; i > 0; i--) {
            var j = Math.floor(Math.random() * (i + 1));
            var temp = array[i];
            array[i] = array[j];
            array[j] = temp;
        }
    }
```

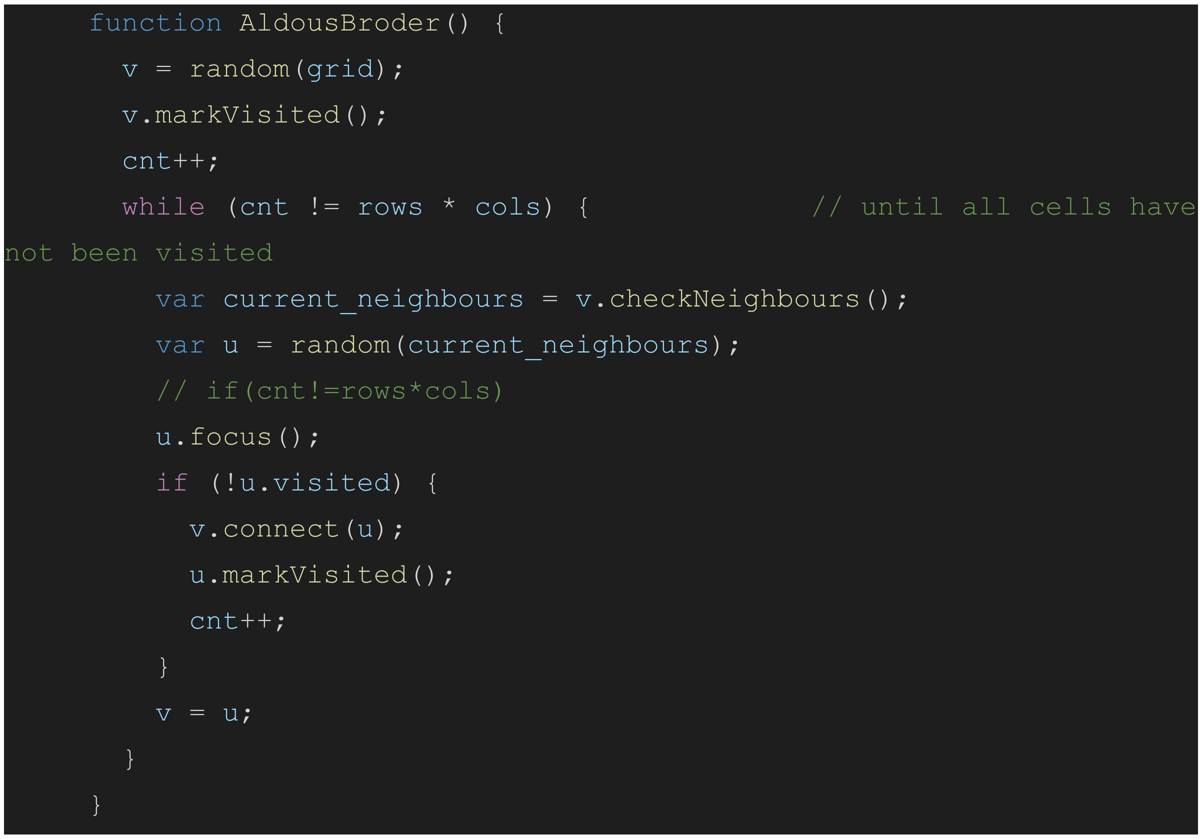
Recursive Backtracking

```
     function  RecursiveBacktracking()    //  implemented  with  explicit
stack
     {
       var start=grid[0];
       start.markVisited();
       stack.push(start);
       start.parent=0;
       while(stack.length>0)
       {
         v=stack.pop();
         var current_neighbours=v.checkNeighbours();
         var u=random(current_neighbours);
         if(u)
         {
           u.markVisited();
           v.connect(u);
           u.parent=v;
           stack.push(u);
         }
         else if(v.parent!=0)
         {
           stack.push(v.parent);
           v.parent.focus();
         }
       }
     }
```

```
     function   RecursiveBacktracking(cell,parent)    //    implemented
recursively without explicit stack
     {
       // visit cell
       cell.markVisited();
       parent.connect(cell);

       // collect neighbours in random order
```

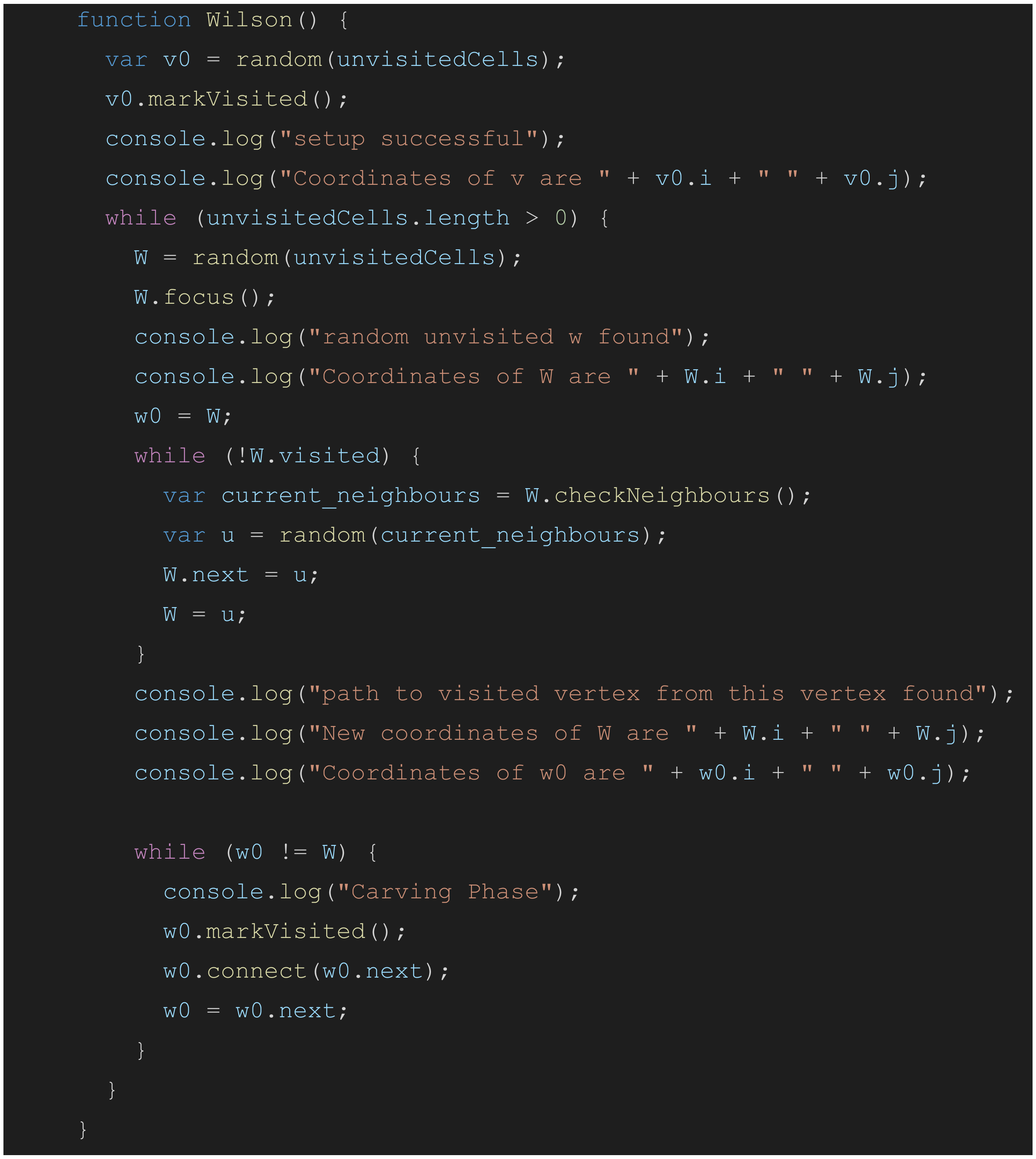
```
    var neighbours=cell.checkNeighbours();
    shuffleArray(neighbours);

    // for neighbours of i
    for(var i=0;i<neighbours.length;i++)
    {
      // if neighbour is not visited
      if(!neighbours[i].visited)
      {
        // DFS(neighbour)
          RecursiveBacktracking(neighbours[i],cell);
      }
    }
  }
```

Aldous Broder



```
     function AldousBroder() {
       v = random(grid);
       v.markVisited();
       cnt++;
       while (cnt != rows * cols) {             // until all cells have
not been visited
         var current_neighbours = v.checkNeighbours();
         var u = random(current_neighbours);
         // if(cnt!=rows*cols)
         u.focus();
         if (!u.visited) {
           v.connect(u);
           u.markVisited();
           cnt++;
         }
         v = u;
       }
     }
```

Wilson Algorithm



```
function Wilson() {
  var v0 = random(unvisitedCells);
  v0.markVisited();
  console.log("setup successful");
  console.log("Coordinates of v are " + v0.i + " " + v0.j);
  while (unvisitedCells.length > 0) {
    W = random(unvisitedCells);
    W.focus();
    console.log("random unvisited w found");
    console.log("Coordinates of W are " + W.i + " " + W.j);
    w0 = W;
    while (!W.visited) {
      var current_neighbours = W.checkNeighbours();
      var u = random(current_neighbours);
      W.next = u;
      W = u;
    }
    console.log("path to visited vertex from this vertex found");
    console.log("New coordinates of W are " + W.i + " " + W.j);
    console.log("Coordinates of w0 are " + w0.i + " " + w0.j);

    while (w0 != W) {
      console.log("Carving Phase");
      w0.markVisited();
      w0.connect(w0.next);
      w0 = w0.next;
    }
  }
}
```

Aldous Broder and Wilson Combined

```
    function Combined() {
       var v0 = Random(unvisitedCells);
       v0.markVisitedUltimate();
       cnt++;
        while (cnt <= 0.3*cols*cols) {               // until 30% cells have
not been visited
           var current_neighbours = v0.checkNeighboursUltimate();
           var u = Random(current_neighbours);
           if (!u.visited) {
               v0.connect(u);
               u.markVisitedUltimate();
               cnt++;
           }
           v0 = u;
       }
       v0=Random(unvisitedCells);
       while (unvisitedCells.length > 0) {
           W = Random(unvisitedCells);
           w0 = W;
           while (!W.visited) {
               var current_neighbours = W.checkNeighboursUltimate();
               var u = Random(current_neighbours);
               W.next = u;
               W = u;
           }
           while (w0 != W) {
               w0.markVisitedUltimate();
               w0.connect(w0.next);
               w0 = w0.next;
           }
       }
    }
```

Hunt and Kill Algorithm

```
function HuntAndKill()
{
  v=grid[0];
  v.markVisited();
  var current_neighbours = v.checkNeighbours();
  while (current_neighbours.length > 0) {
    var u = random(current_neighbours);
    u.markVisited();
    v.connect(u);
    v = u;
    var current_neighbours = v.checkNeighbours();
  }
  for (var i = 0; i < grid.length; i++) {
    var cell = grid[i];
    cell.focus();
    var adjacent = cell.checkVisitedNeighbour();
    if (!cell.visited && (adjacent)) {
      v = cell;
      v.connect(adjacent);
      // adjacent.connect(v);
      v.markVisited();
      var current_neighbours = v.checkNeighbours();
      while (current_neighbours.length > 0) {
        var u = random(current_neighbours);
        u.markVisited();
        v.connect(u);
        v = u;
        var current_neighbours = v.checkNeighbours();
      }
    }
  }
}
```

```
class Vertex:
    def __init__(self, n, i,j,walls):
        self.index = n
        self.i=i
        self.j=j
        self.neighbors = list()
        self.visited=False
        self.parent=-1
        self.walls=walls
    def add_neighbor(self, v):
        self.neighbors.append(v)
        # self.neighbors.sort()
    def is_dead_end(self):
        if len(self.neighbors)==1:
            return True
        return False
    def is_intersection(self):
        if len(self.neighbors)>2:
            return True
        return False

class Graph:

    def __init__(self):
        self.vertices = {}
        self.cols=0

        self.dfs_flag=True
        self.dfs_cnt=-1
        self.dfs_cnt_intersection=0
        self.dfs_cnt_dead_end=0
        self.dfs_traversal=[]

        self.bfs_flag=True
        self.bfs_cnt=-1
        self.bfs_cnt_intersection=0
        self.bfs_cnt_dead_end=0
        self.bfs_traversal=[]

        self.dfs_heuristics_flag=True
        self.dfs_heuristics_cnt=-1
        self.dfs_heuristics_cnt_intersection=0
        self.dfs_heuristics_cnt_dead_end=0
        self.dfs_heuristics_traversal=[]

        self.random_walk_flag=True
        self.random_walk_cnt=-1
        self.random_walk_cnt_intersection=0
        self.random_walk_cnt_dead_end=0
        self.random_walk_traversal=[]

    def neutralize(self):
        for i in self.vertices.keys():
            self.vertices[i].visited=False

    def add_vertex(self, vertex):
        self.vertices[vertex.index] = vertex
        return True

    def add_edge(self, u, v):
        self.vertices[u].add_neighbor(v)
        self.vertices[v].add_neighbor(u)
        return True

    def construct_graph(self):
        for vertex in self.vertices.values():
            # vertex=self.vertices[key]
            if(not vertex.walls[1]):
                self.add_edge(vertex.index, vertex.index+1)
            if(not vertex.walls[2]):
                self.add_edge(vertex.index, vertex.index+self.cols)
    def cnt_dead_ends(self):
        cnt=0
        for i in self.vertices.values():
            if i.is_dead_end():
                cnt+=1
        return cnt

    def cnt_intersections(self):
        cnt=0
        for i in self.vertices.values():
            if i.is_intersection():
                cnt+=1
        return cnt

    def direction(self,v,parent):
        value = self.vertices[v].index - self.vertices[parent].index;
        if (value == 1):
            return "right"
        if (value == -1):
            return "left"
        if (value == self.cols) :
            return "down"
        if (value == -self.cols):
            return "up"

    def dfs(self,v,destination):
        #  visit v
        self.dfs_cnt+=1
        self.vertices[v].visited=True
        if self.vertices[v].is_intersection():
            self.dfs_cnt_intersection+=1
        if self.vertices[v].is_dead_end():
            self.dfs_cnt_dead_end+=1
        # self.vertices[v].parent=parent
        self.dfs_traversal.append(v)
        # print(self.direction(v,parent))
        if v==destination:
            self.dfs_flag=False
            print("Reached Destination DFS")
            print("No of steps need = ",self.dfs_cnt)
                              print("No  of  intersections  visited  =
",self.dfs_cnt_intersection)
            print("No of dead ends visited = ",self.dfs_cnt_dead_end)


        for i in self.vertices[v].neighbors:
            if (not self.vertices[i].visited) and self.dfs_flag:
                self.dfs(i,destination)

    def bfs(self,v,destination):
        queue=deque()
        queue.append(v)
        while(len(queue) and self.bfs_flag):
            u=queue.popleft()
            self.bfs_cnt+=1
            self.vertices[u].visited=True
            if self.vertices[u].is_intersection():
                self.bfs_cnt_intersection+=1
            if self.vertices[u].is_dead_end():
                self.bfs_cnt_dead_end+=1
            # self.vertices[u].parent=parent
            self.bfs_traversal.append(u)
            if u==destination:
                self.bfs_flag=False
                print("Reached Destination BFS")
                print("No of steps need = ",self.bfs_cnt)
                                print("No  of  intersections  visited  =
",self.bfs_cnt_intersection)
                                  print("No  of  dead  ends  visited  =
",self.bfs_cnt_dead_end)
            else:
                for i in self.vertices[u].neighbors:
                    if (not self.vertices[i].visited):
```

```
              queue.append(i)

    def dfs_heuristics(self,v,destination):
      self.dfs_heuristics_cnt+=1
      self.vertices[v].visited=True
      if self.vertices[v].is_intersection():
        self.dfs_heuristics_cnt_intersection+=1
      if self.vertices[v].is_dead_end():
        self.dfs_heuristics_cnt_dead_end+=1
      # self.vertices[v].parent=parent
      self.dfs_heuristics_traversal.append(v)
      # print(self.direction(v,parent))
      if v==destination:
        self.dfs_heuristics_flag=False
        print("Reached Destination DFS_heuristics")
        print("No of steps need = ",self.dfs_heuristics_cnt)
                 print("No   of   intersections   visited   =
",self.dfs_heuristics_cnt_intersection)
                  print("No   of   dead   ends   visited   =
",self.dfs_heuristics_cnt_dead_end)

                             for      i      in
list(self.order_heuristics(self.vertices[v].neighbors,destination)):
                 if   (not   self.vertices[i].visited)   and
self.dfs_heuristics_flag:
          self.dfs_heuristics(i,destination)

    def random_walk(self,v,destination):
      while(v!=destination):
          self.random_walk_cnt+=1
          self.vertices[v].visited=True
          if self.vertices[v].is_intersection():
            self.random_walk_cnt_intersection+=1
          if self.vertices[v].is_dead_end():
            self.random_walk_cnt_dead_end+=1
          self.random_walk_traversal.append(v)
          v=random.choice(self.vertices[v].neighbors)
        print("Reached Destination Random Walk")
        print("No of steps need = ",self.random_walk_cnt)
                                print("No of intersections visited =
",self.random_walk_cnt_intersection)
                                 print("No of dead ends visited =
",self.random_walk_cnt_dead_end)

    def order_heuristics(self,l,destination):
        array=[]
        for i in l:
            array.append([self.manhattan(i,destination),i])
        array.sort()
        l=[]
        for i in array:
            l.append(i[1])
        return l

    def manhattan(self,i,d):
        x1=self.vertices[i].i
        y1=self.vertices[i].j
        x2=self.vertices[d].i
        y2=self.vertices[d].j
        return abs(x1-x2)+abs(y1-y2)

    def print_graph(self):
        for key in sorted(list(self.vertices.keys())):
            print(str(key)+" : " + str(self.vertices[key].neighbors))
```