PROJECT REPORT

Maze Generator & Solver

Submitted in partial fulfilment of the requirements

for the award of the degree of

Bachelor of Technology in

Electronics and Communication

Submitted by:

Harshit Rajput (2K19/EC/072) &

Jangid Abhishek Vijay (2K19/EC/080)

Under the supervision of

Assistant Prof. Rohit Kumar

DELHI TECHNOLOGICAL UNIVERSITY

Formerly known as, Delhi College of Engineering

Bawana Road, Delhi-110042

CANDIDATE'S DECLARATION

*We, HARSHIT RAJPUT AND JANGID ABHISHEK VIJAY, Roll No(s). 2K19/EC/072 and 2K19/EC/080 student(s) of B. Tech. ELECTRONICS AND COMMNICATION, hereby declare that the project Dissertation titled "**Maze Generator & Solver**" which is submitted by us to the Department of electronics and communication Delhi Technological University, Delhi in partial fulfilment of the requirement for the award of the degree Bachelor of Technology, is original and not copied from any source without proper citation. This work has not previously formed the basis for the award of any Degree, Diploma Associateship, Fellowship or other similar title or recognition.*

Place: Delhi

HARSHIT RAJPUT & JANGID ABHISHEK VIJAY

DATE:26.11.2021

CERTIFICATE

I hereby certify that the Project Dissertation titled " ***Maze Generator & Solver****" which is submitted by* ***HARSHIT RAJPUT AND JANGID ABHISHEK VIJAY Roll No(s). 2K19/EC/072 AND 2K19/EC/080*** *of BTech. Electronic andcommunication, Delhi Technological University, Delhi in partial fulfilment of therequirement for the award of Bachelor of Technology, is a record of the project work carried out by the students under my supervision. To the best of my knowledge this work has not been submitted in part or full for any Degree or Diploma to this University or elsewhere.*

Place: Delhi *Prof. Rohit Kumar*

Date:26.11.2021

ACKNOWLWDGEMENT

We would like to express my special thanks of gratitude to my teacher Prof. Rohit Kumar and Mr. Vishal Kumar, who gave us the golden opportunity to do this wonderful project of Algorithm design & analysis.

Who also helped me in completing my project! We came to know about so many new things I am really thankful to them. Secondly, we would also like to thank my parents and friends who helped me a lot in finalizing the project within the limited time frame.

List of Contents

1. Introduction

This project has two algorithmic parts to it. The first one is about generating the 20x20 grid sized maze and second one is about solving it. There are a total of 400 cells named from 0 to 399. The first row contains the cells from 0 to 19. The second row contains cells from 20 to 39 and so on.

The language we have used over here for the complete program is python, as it is easy to understand. In order to visualize all this, we have used pygame as our animation software.

The generation of maze is done by the Recursive deep first search algorithm. While carving the maze itself, a map stores the parent of all vertices as we proceed while carving the maze. Hence, we have the shortest path ready with us by the time whole maze is carved out.

As an additional solution to it using the very famous Dijkstra's algorithm, we have created that function in such a way that it takes two arguments in it. One is the source vertex and other is destination vertex. So, the shortest possible route from the source vertex to the destination vertex will be plotted.

Note: The code link is given here. This complete code is unique and is not exactly copied from anywhere. Many additional features and functions are added to the code and will be also developed further in future.

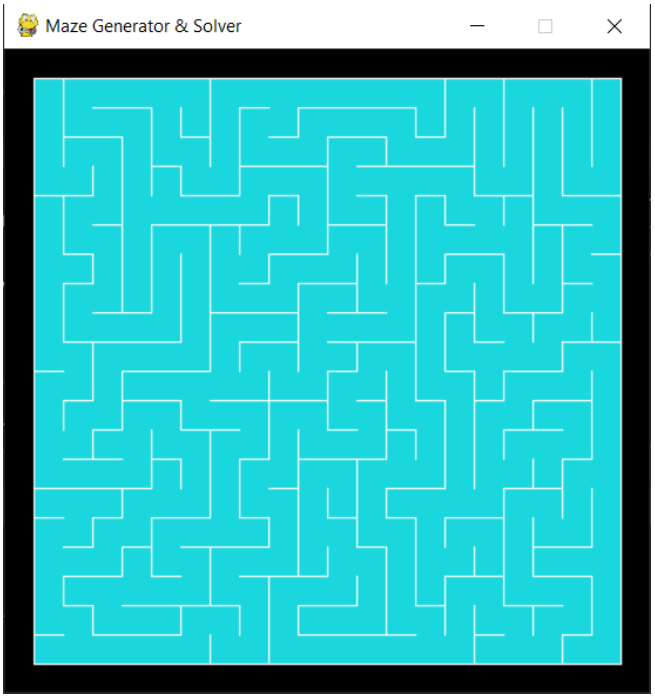


Figure 1. The visual representation of generation of 20x20 maze carved from a 20x20 grid.

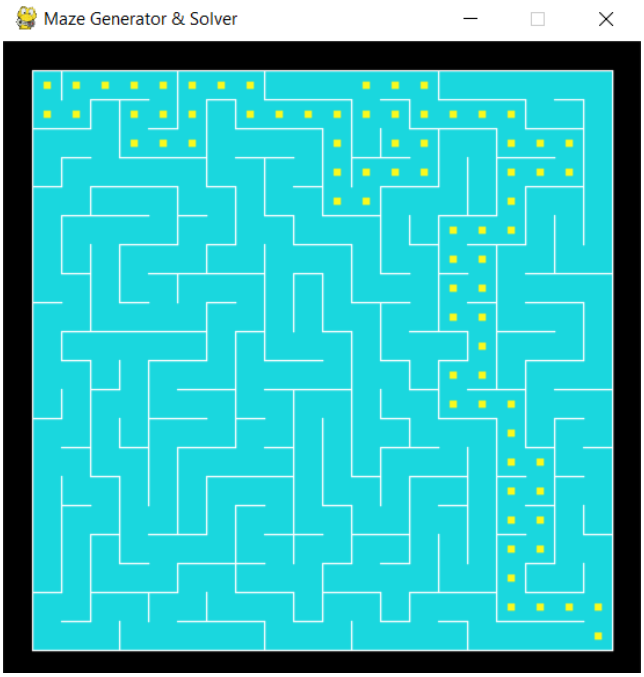


Figure 2. The shortest possible path plotted using the map that we created while carving the maze.

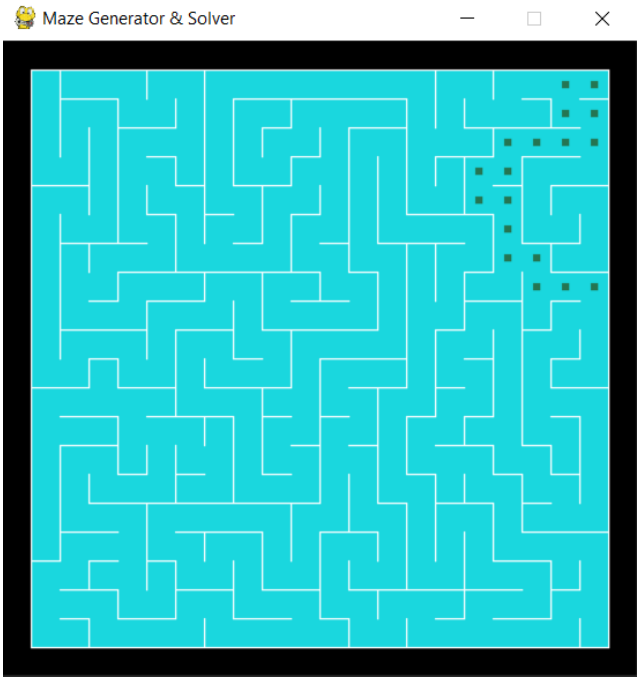


Figure 3. The maze solved using Dijkstra's Algorithm for the source vertex as 19 (row 0 and column 19) and destination vertex as 159 (row 8 and column 19).

2. Problems faced

We had no clear idea that how a 20x20 maze can be put into the program such that computer can read that out. There was some intuition that graph as the data structure could be used to represent this maze, but we were not able to figure it out.

The foremost challenge was how to represent the maze with the help of graph? The next challenge was representing the project with the help of visualizer. Pygame was quite new for us to understand. The arguments given to the functions in pygame is usually in the unit of pixels. So, it became quite obvious to make functions like converting the pixel number to the corresponding vertex or cell number and vice versa.

```
import copy

import pygame
import time
import random

# set up pygame window
WIDTH = 440
HEIGHT = 440
FPS = 30

# Define colours
WHITE = (255, 255, 255)
GREEN = (37, 117, 82,)
BLUE = (26, 215, 222)
YELLOW = (249, 249, 30)

# initialise Pygame
pygame.init()
pygame.mixer.init()
screen = pygame.display.set_mode((WIDTH, HEIGHT))
pygame.display.set_caption("Maze Generator & Solver")
clock = pygame.time.Clock()
```

Figure 4. shows the initial pygame setup (like colour selection and describing the size of cells) before creating the maze.

3. Maze Generation

For the generation of maze, we have used the Recursive deep first search algorithm. Recursive deep first search algorithm goes as follows:

1. Given a current cell as a parameter,
2. Mark the current cell as visited
3. While the current cell has any unvisited neighbor cells
    1. Choose one of the unvisited neighbors
    2. Remove the wall between the current cell and the chosen cell
    3. Invoke the routine recursively for a chosen cell

How the maze is represented in the form of graph?

To represent the maze in pygame visualizer, we will first build the 20x20 grid with the help of white lines and then carve the maze out of it using the ocean blue color. To represent the maze in the form of graph surely, we will need an adjacency matrix. Since it is a 20x20 maze, there would be 400 cells and to show the connections of these 400 cells we would require a 400x400 matrix.

We will also require a 20x20 bool matrix to keep a track of visited cells. Along with that, we will also use a stack to keep the history of cells we came across. A HashMap would be required to store the parent of respective cells in which the shortest path would be stored.

```
33  # setup maze variables
34  x = 0  # x axis
35  y = 0  # y axis
36  w = 20  # width of cell
37
38  row_col_size = 20
39  size = row_col_size * row_col_size
40  adjacencey_mat = [[0 for i in range(size)] for j in range(size)]
41
42  grid = []
43  visited = []
44  stack = []
45  solution = {}
```

Figure 5. explains how the above-mentioned data structures are set up in python code.

```
262  # main function call ----------------------------------------------------------------------------------------------------
263  x, y = 20, 20              # starting position of grid
264  build_grid(40, 0, 20)      # 1st argument = x value, 2nd argument = y value, 3rd argument = width of cell
265  carve_out_maze(x, y)       # call build the maze  function
266  plot_route_back(400, 400)  # call the plot solution function
267  dijkstra(19, 159)
```

Figure 6. shows the main function of our program where in, 4 functions are called one by one.

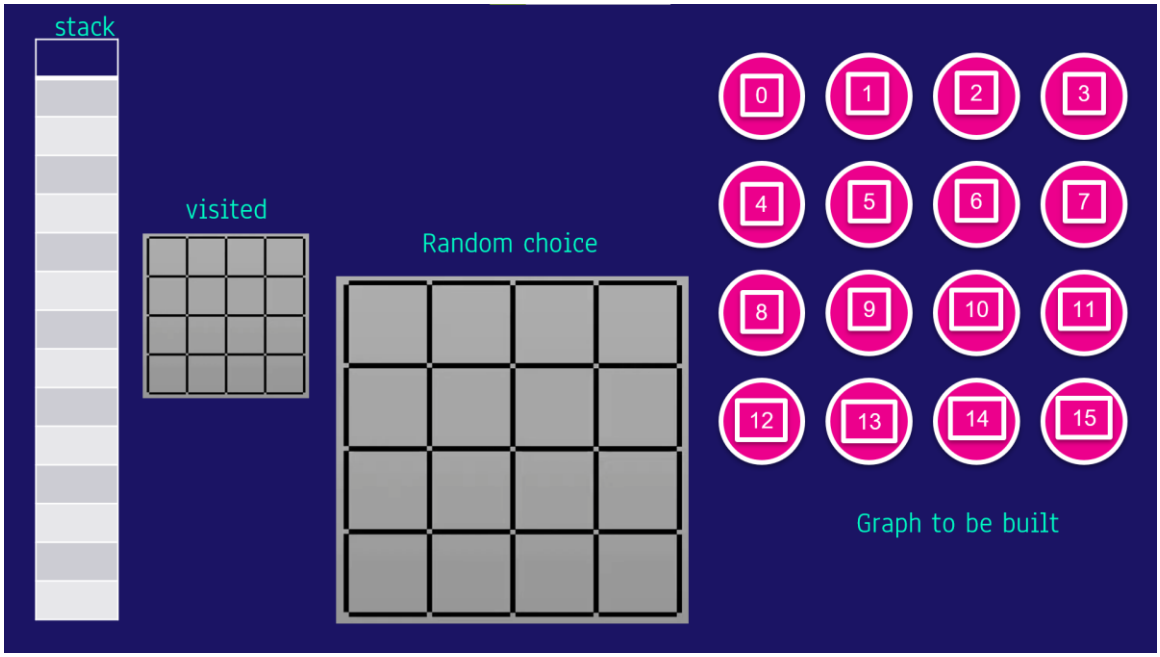
The first function buil_grid (40, 0, 20) will make the 20x20 grids using white lines.

The second function carve_out_maze (x, y) will carve the maze from the grid made using function 1 as shown in figure 1.

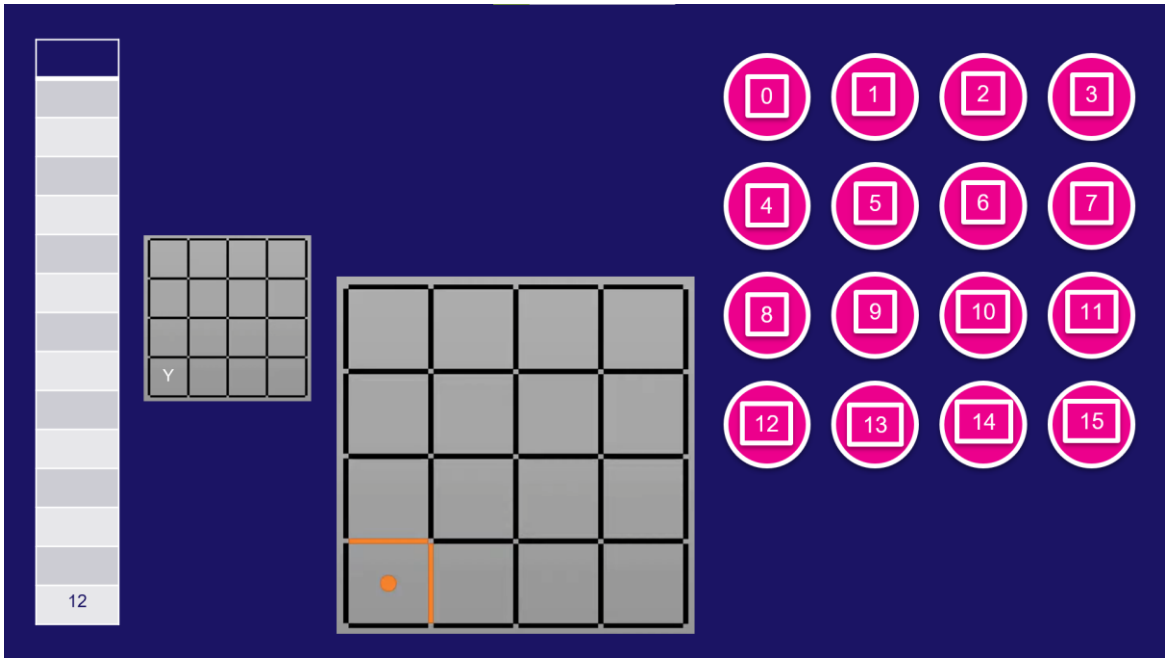
The third function plot_route_back (400, 400) will plot the shortest possible path from top-left corner to bottom-right corner using the map that we store while carving the maze as shown in figure 2.

The fourth function dijkstra (19, 159) will plot the shortest possible path from cell 19 to 159 as shown in figure 3.

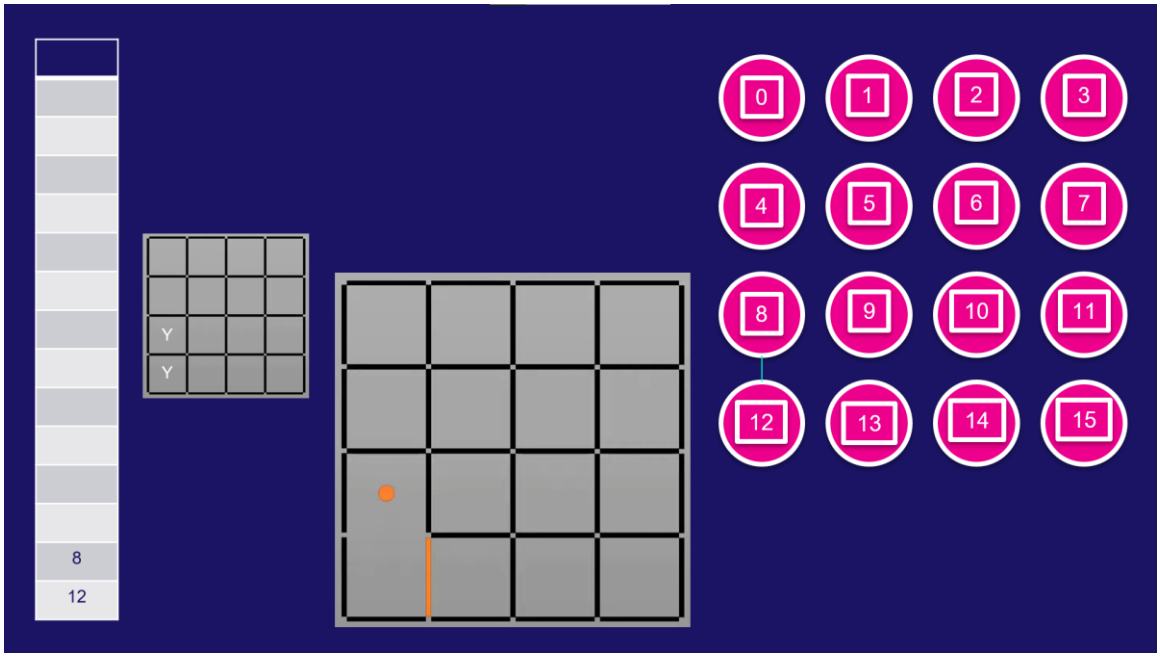
Let us understand the maze generation on a smaller 4x4 grid.



The “Graph to be built” contains total 16 cells numbered from 0 to 15. And as mentioned before, we have a stack and a bool matrix named as “visited”.



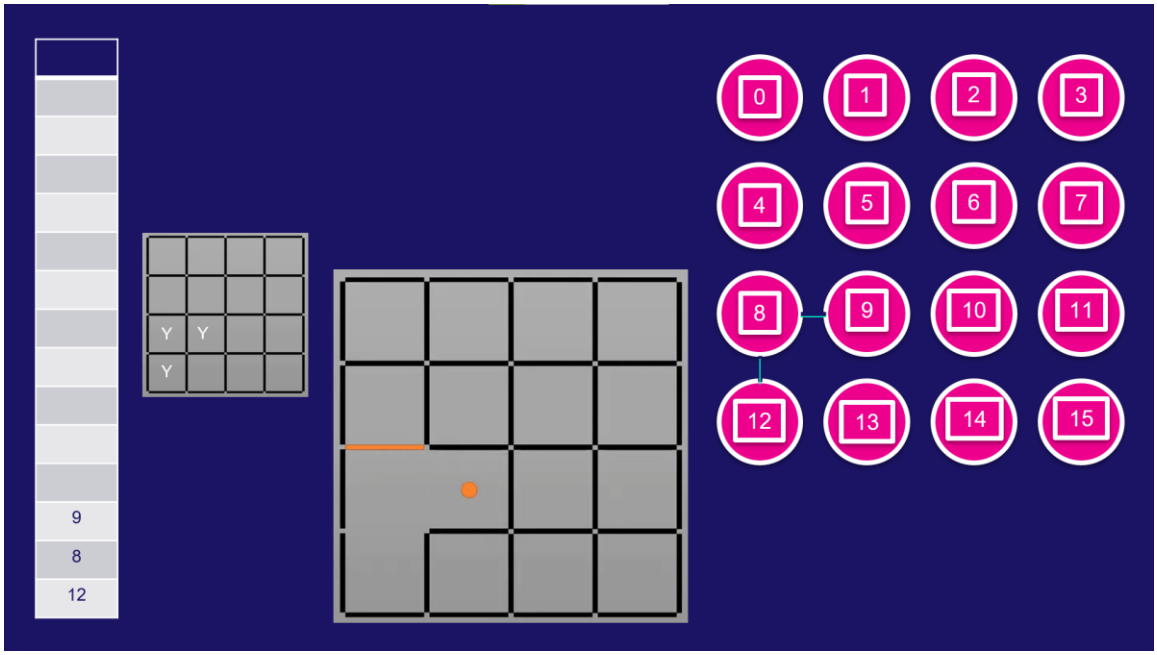
Suppose, we start from the cell 12. So, let’s mark the same cell as visited and also add it to the stack. Here, the pointer has two available neighbours. One is 8 (top) and other is 13 (right).



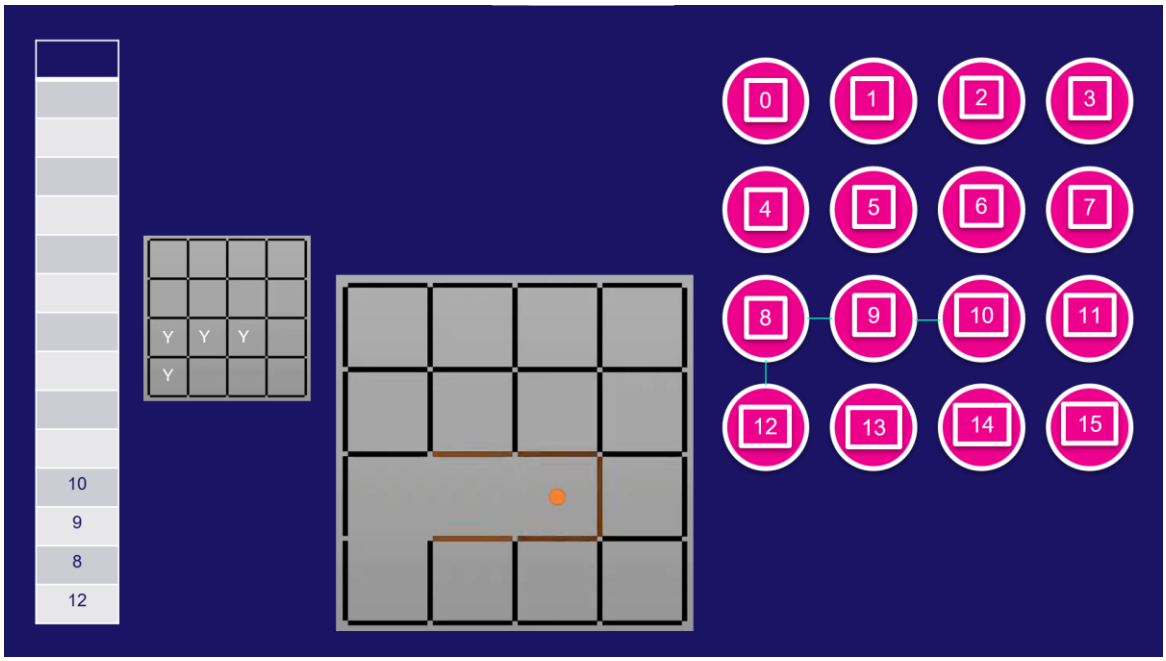
Let's say that cell selects to go to its top neighbour. So, we will mark cell 8 as visited and will add it to the stack. Further, in the solution dictionary (HashMap) 12 is marked as parent of 8. So, the cell 12 and 8 are now connected to each other (i.e., adjacenecy_matrix[12][8] and adjacenecy_matrix[8][12] are given the weights 1).

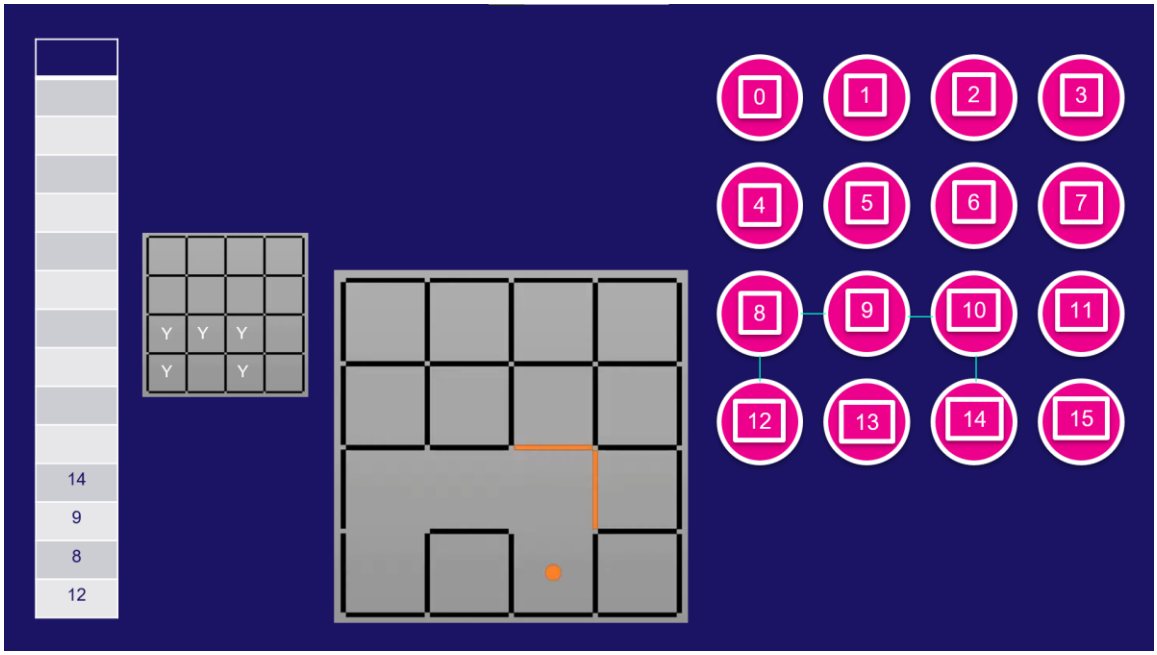
To represent this in our visualizer, rather than removing the wall between cell 8 and 12, we will overwrite the grid with blue colour to make them look like the wall has been removed.

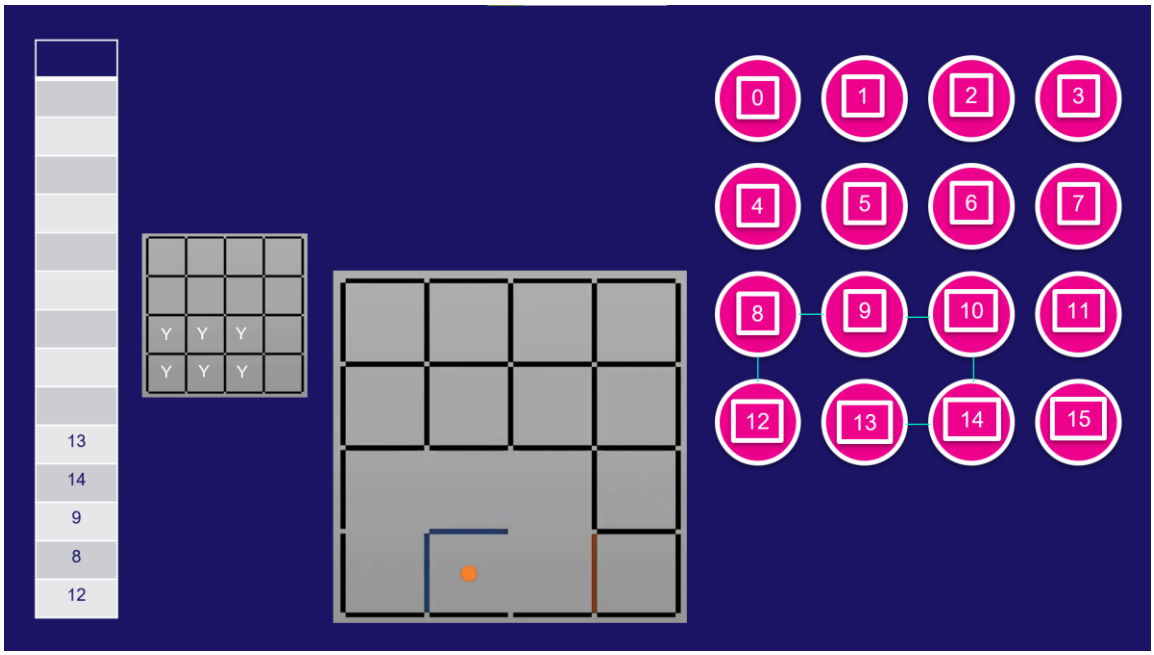
Similarly, now the cell has again 2 options to move upwards or rightwards.



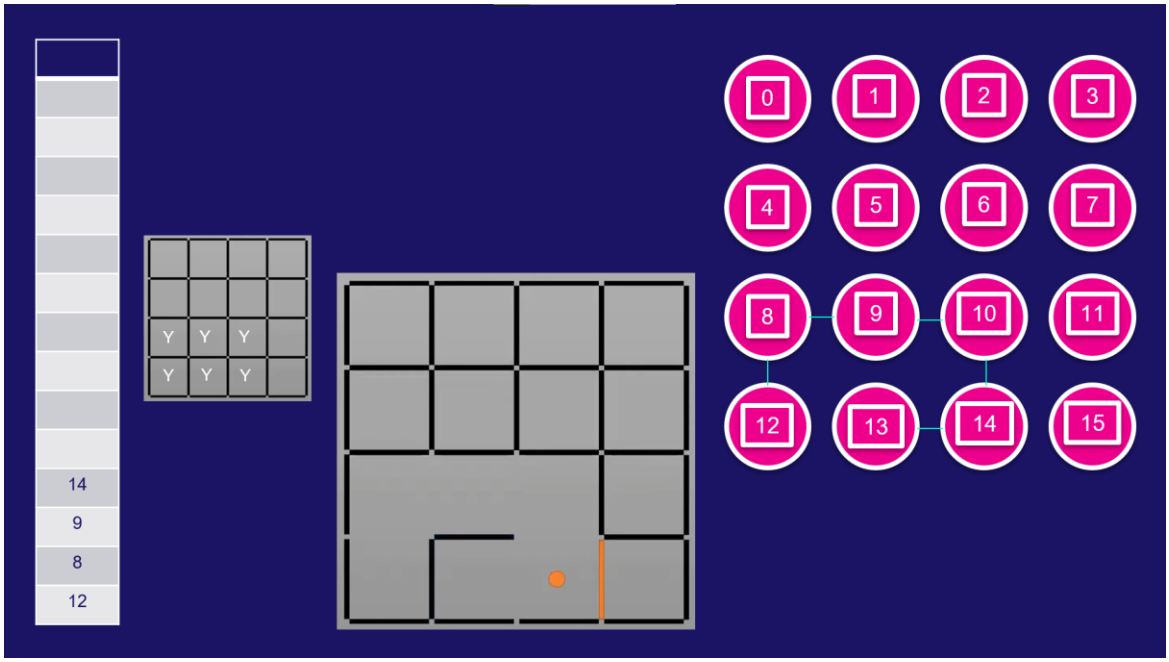
Suppose the cell moves rightwards. So, the same process repeats and the connections in the graph are respectively made.

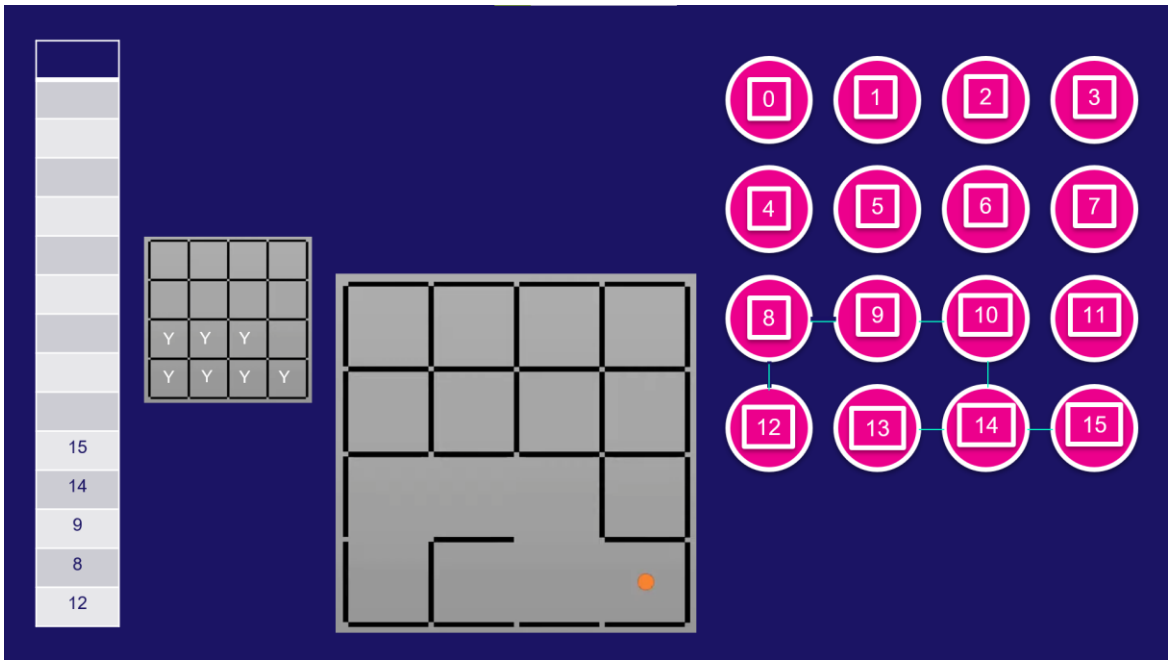


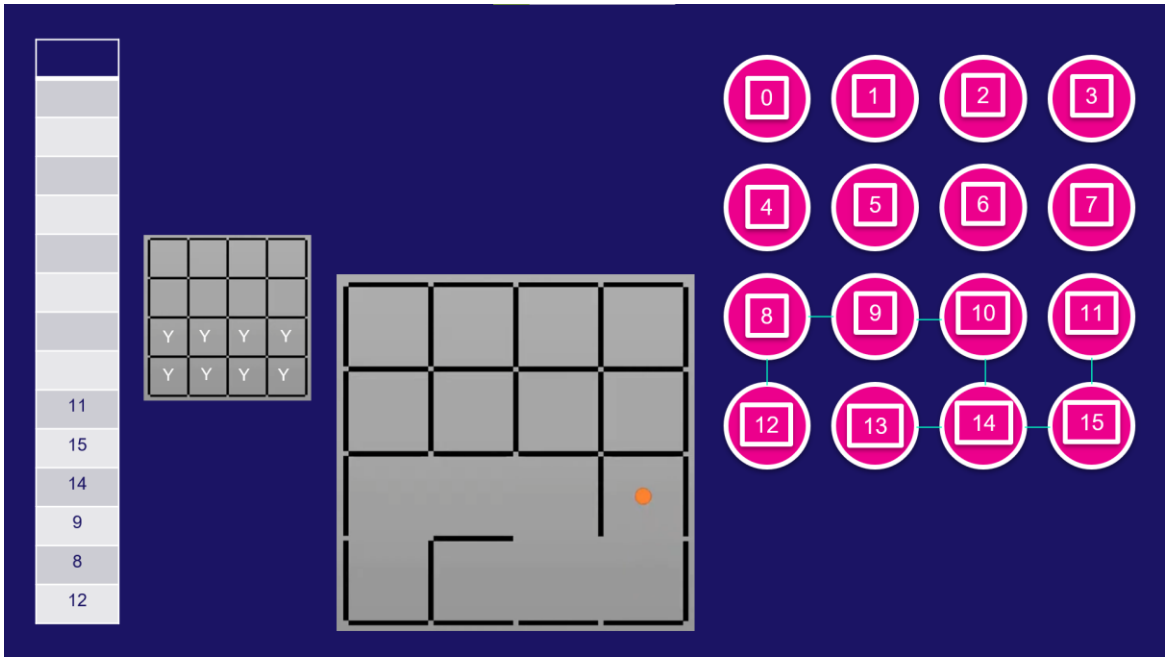


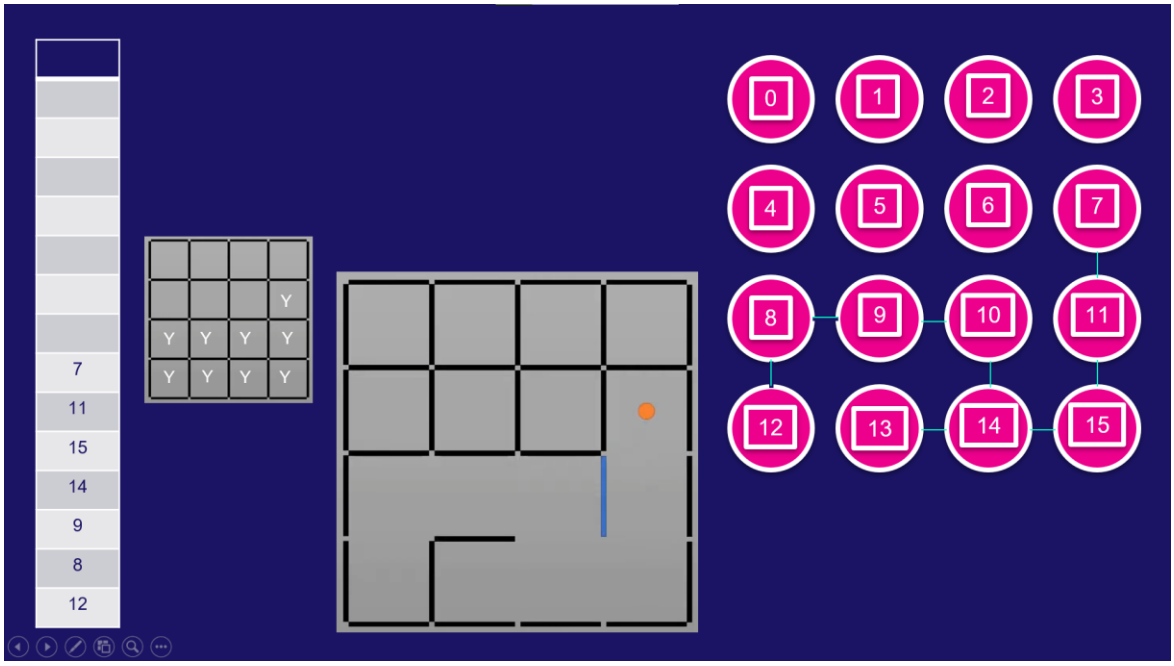


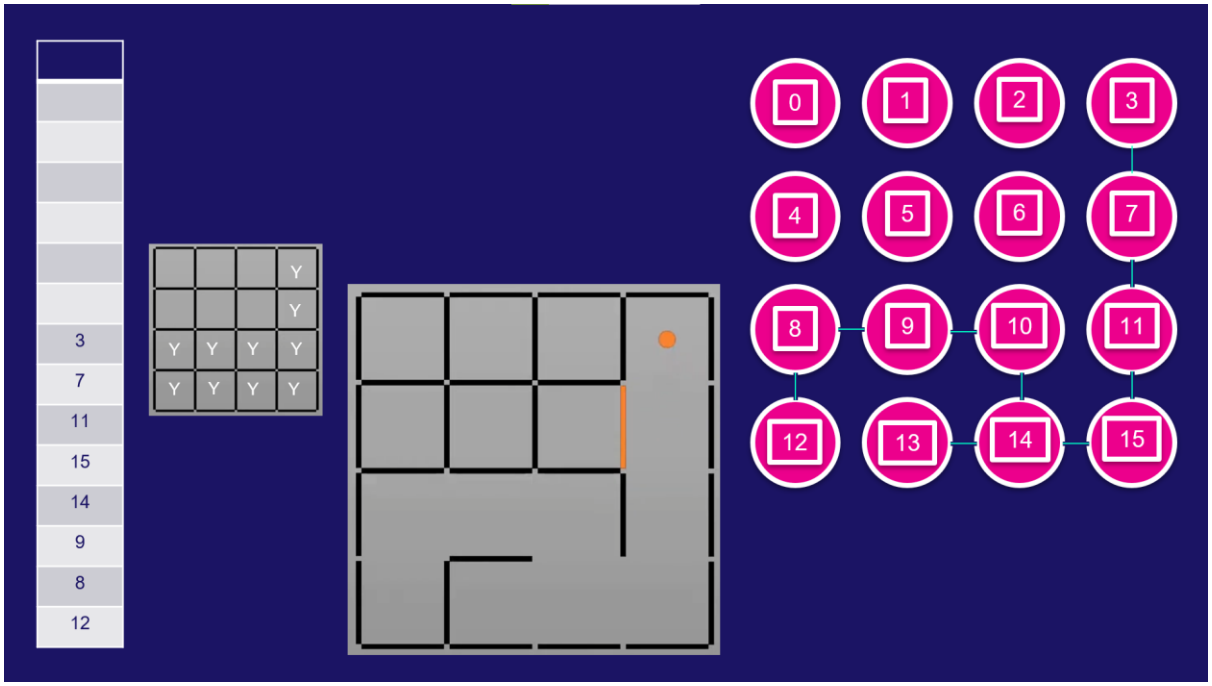
Now, here the cell has no further option to move because all the neighbour cells are already visited. So, using the stack the pointer will now backtrack to the previous cell and look for the unvisited neighbour cells.

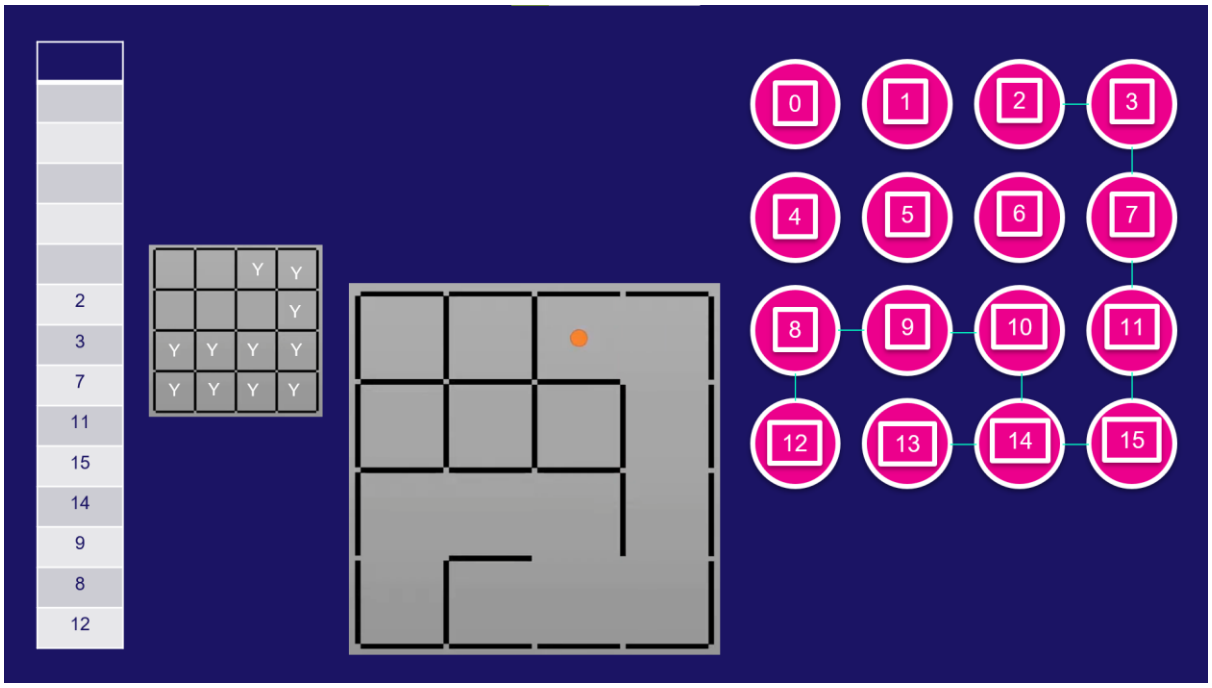

14
9
8
12
Y Y Y
Y Y Y
0 1 2 3
4 5 6 7
8 9 10 11
12 13 14 15

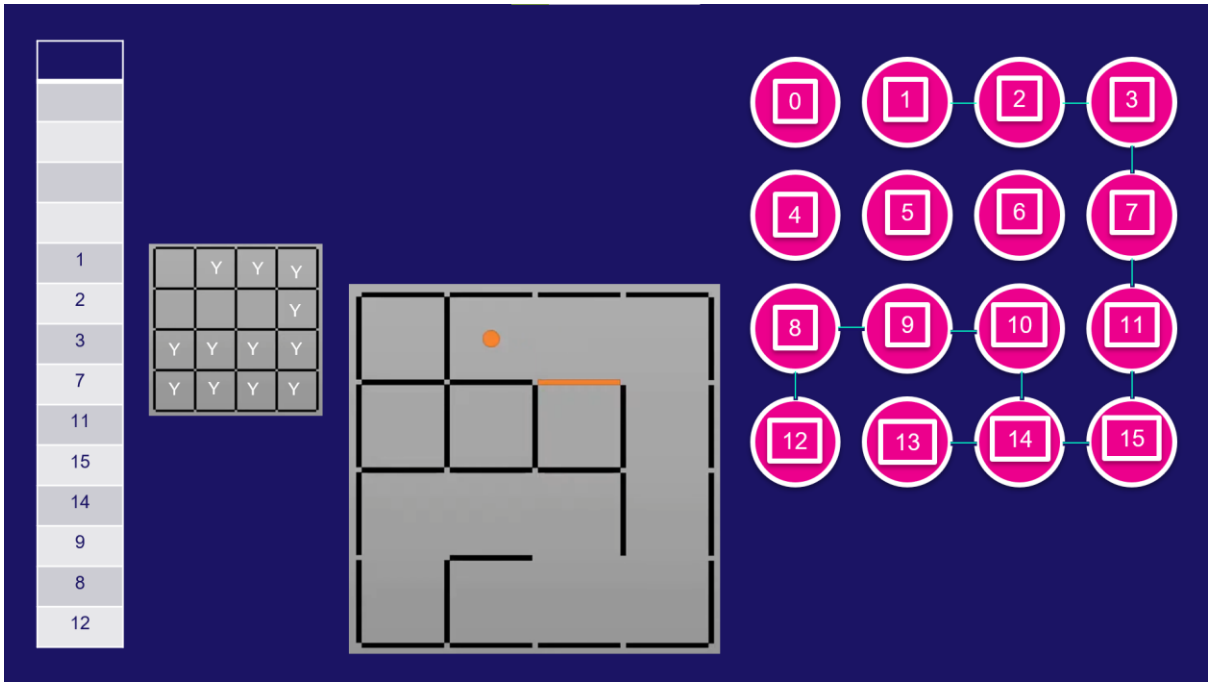

15
14
9
8
12
Y Y Y
Y Y Y Y
0 1 2 3
4 5 6 7
8 9 10 11
12 13 14 15

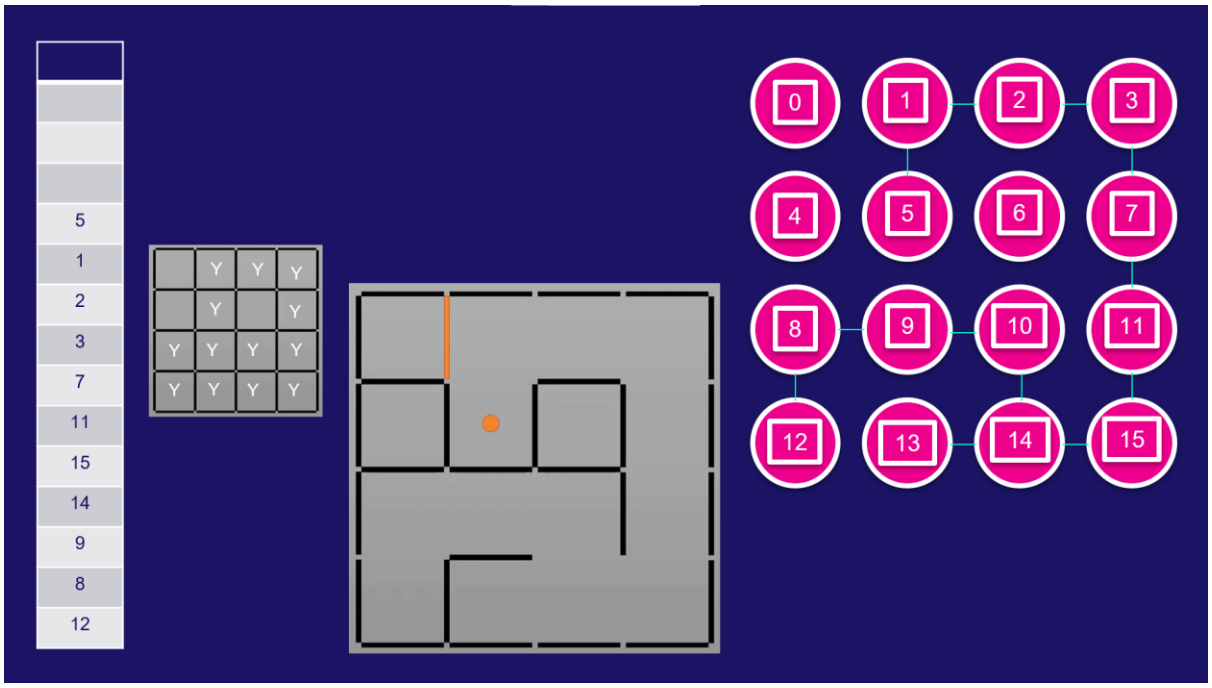

11
15
14
9
8
12
Y Y Y Y
Y Y Y Y
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

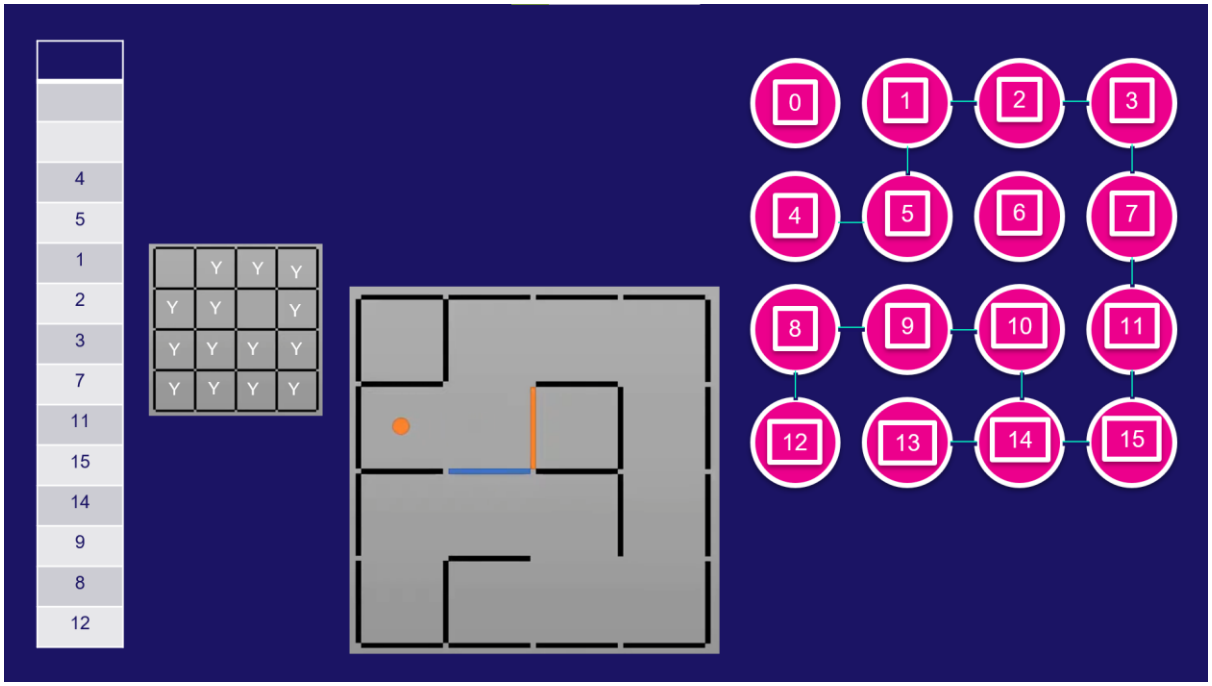

7
11
15
14
9
8
12
Y
Y Y Y Y
Y Y Y Y
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

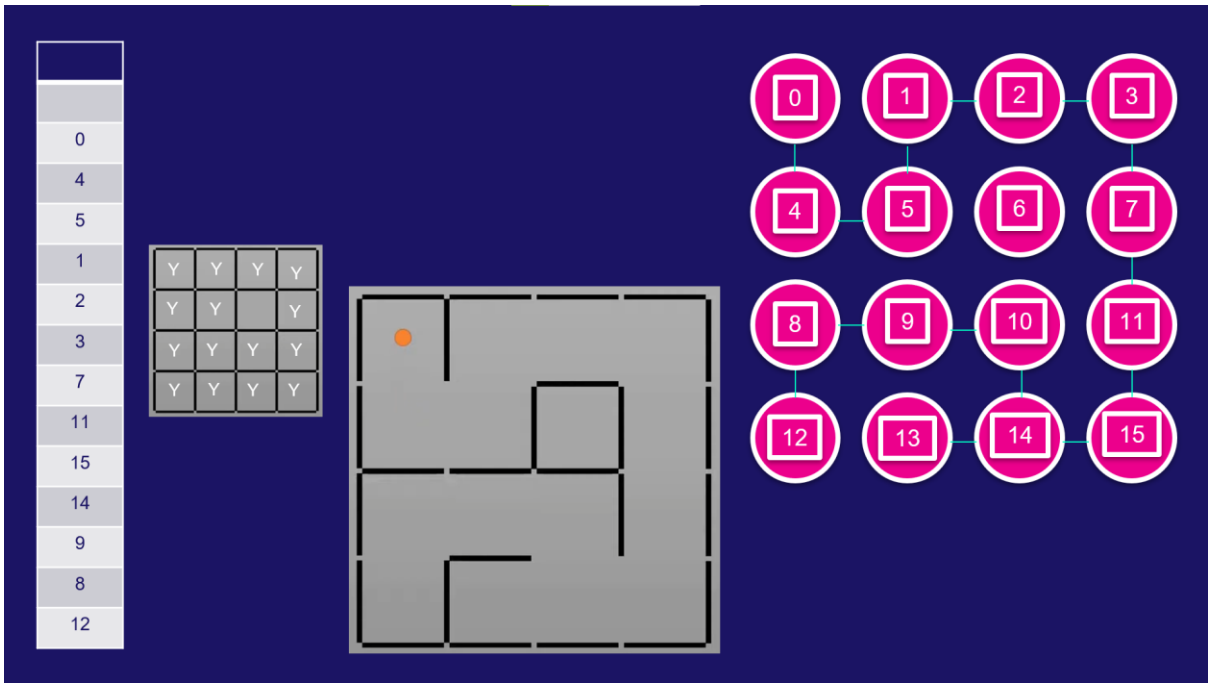

3
7
11
15
14
9
8
12
Y
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

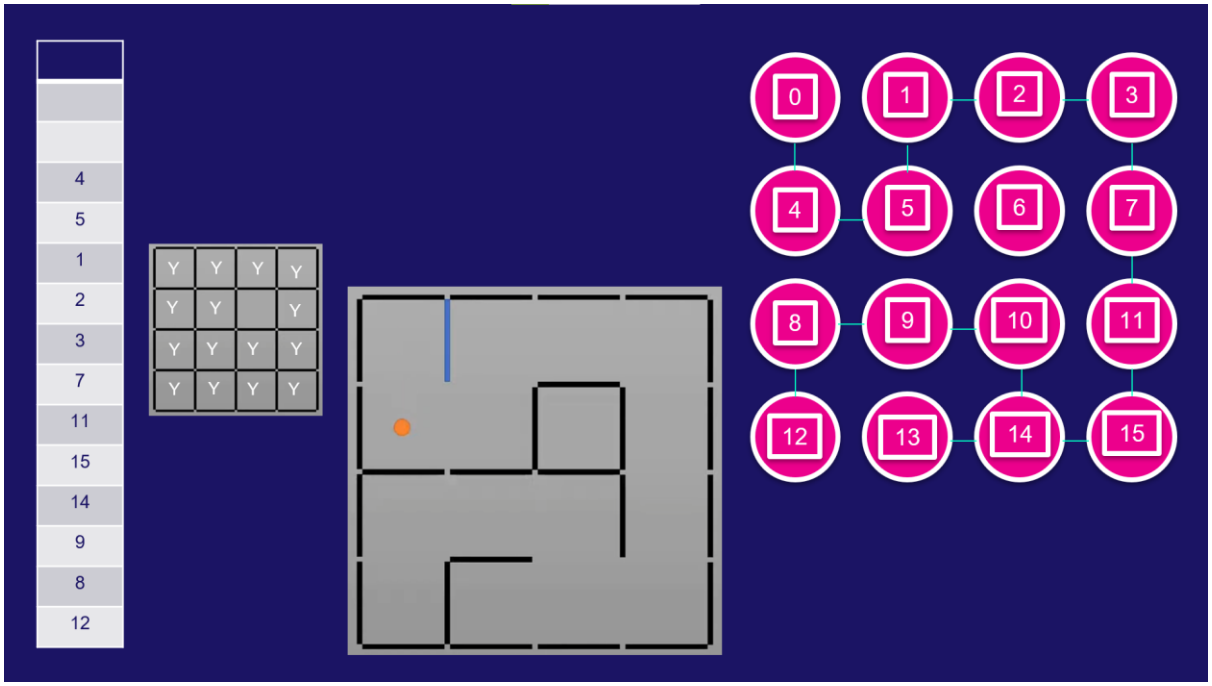

2
3
7
11
15
14
9
8
12
Y
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

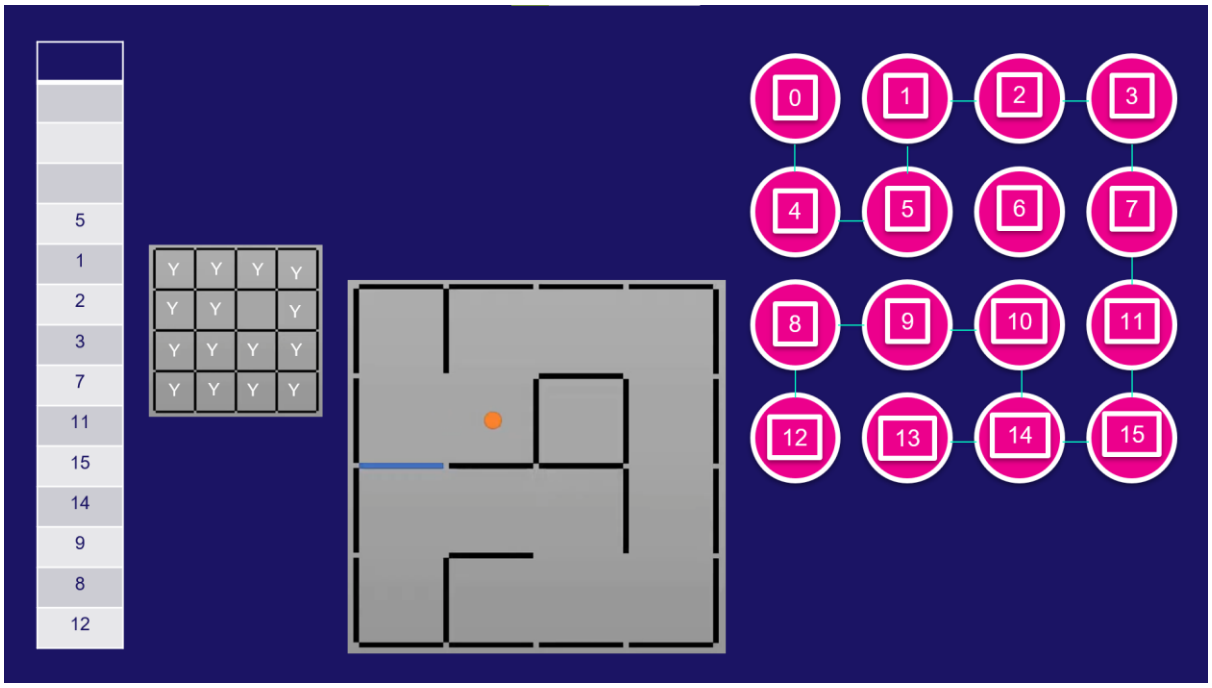

1
2
3
7
11
15
14
9
8
12
Y
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

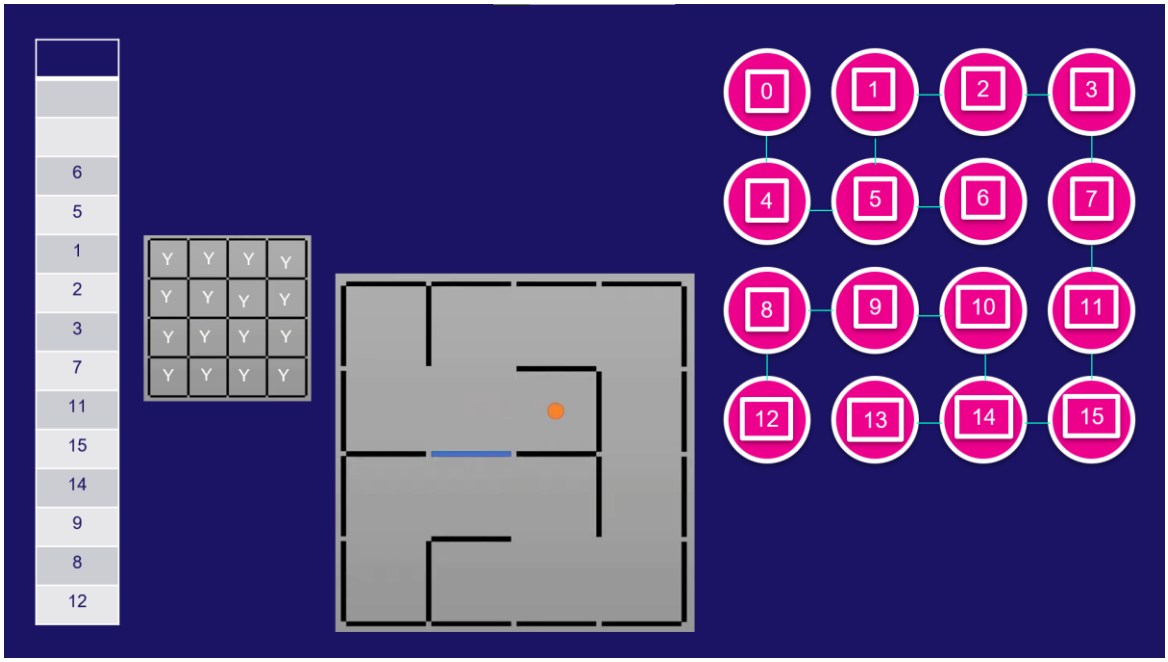

5
1
2
3
7
11
15
14
9
8
12
Y
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15


4
5
1
2
3
7
11
15
14
9
8
12
Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15


0
4
5
1
2
3
7
11
15
14
9
8
12
Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15


4
5
1
2
3
7
11
15
14
9
8
12
Y Y Y Y
Y Y Y
Y Y Y Y
Y Y Y Y
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15


5
1
2
3
7
11
15
14
9
8
12
Y Y Y Y
Y Y Y
Y Y Y Y
Y Y Y Y
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15



Now, all the cells are marked visited which means our graph representation is ready, both in the adjacency matrix form and the pygame visualizer form.

Code in python

```
# build the grid ------------------------------------------------------------------------------------------------------
def build_grid(x, y, w):
    for i in range(1, 21):
        x = 20                                                          # set x coordinate to start position
        y = y + 20                                                      # start a new row
        for j in range(1, 21):
            pygame.draw.line(screen, WHITE, [x, y], [x + w, y])         # top of cell
            pygame.draw.line(screen, WHITE, [x + w, y], [x + w, y + w]) # right of cell
            pygame.draw.line(screen, WHITE, [x + w, y + w], [x, y + w]) # bottom of cell
            pygame.draw.line(screen, WHITE, [x, y + w], [x, y])         # left of cell
            grid.append((x, y))                                         # add cell to grid list
            x = x + 20                                                  # move cell to new position

```

The first function runs and hence the 20x20 grid is generated.

```
127 def carve_out_maze(x, y):
128     single_cell(x, y)                                   # starting positing of maze
129     stack.append((x, y))                                # place starting cell into stack
130     visited.append((x, y))                              # add starting cell to visited list
131
132     while len(stack) > 0:                               # loop until stack is empty
133         time.sleep(.03)                                 # slow program now a bit
134         cell = []                                       # define cell list
135         if (x + w, y) not in visited and (x + w, y) in grid:        # right cell available?
136             cell.append("right")                        # if yes add to cell list
137
138         if (x - w, y) not in visited and (x - w, y) in grid:        # left cell available?
139             cell.append("left")
140
141         if (x, y + w) not in visited and (x, y + w) in grid:        # down cell available?
142             cell.append("down")
143
144         if (x, y - w) not in visited and (x, y - w) in grid:        # up cell available?
145             cell.append("up")
146
147         if len(cell) > 0:                               # check to see if cell list is empty
148             cell_chosen = (random.choice(cell))         # select one of the cell randomly
149
150             p = pixel_to_vertex(x, y)
151             q = pixel_to_vertex(x + w, y)
152             r = pixel_to_vertex(x - w, y)
153             s = pixel_to_vertex(x, y + w)
154             t = pixel_to_vertex(x, y - w)
```

```
154             t = pixel_to_vertex(x, y - w)
155
156             if cell_chosen == "right":                  # if this cell has been chosen
157                 adjacencey_mat[p][q] = 1
158                 adjacencey_mat[q][p] = 1
159
160                 push_right(x, y)                        # call push_right function
161                 solution[(x + w, y)] = x, y             # solution = dictionary key = new cell, other = current cell
162                 x = x + w                               # make this cell the current cell
163                 visited.append((x, y))                  # add to visited list
164                 stack.append((x, y))                    # place current cell on to stack
165
166             elif cell_chosen == "left":
167                 adjacencey_mat[p][r] = 1
168                 adjacencey_mat[r][p] = 1
169
170                 push_left(x, y)
171                 solution[(x - w, y)] = x, y
172                 x = x - w
173                 visited.append((x, y))
174                 stack.append((x, y))
175
176             elif cell_chosen == "down":
177                 adjacencey_mat[p][s] = 1
178                 adjacencey_mat[s][p] = 1
179
180                 push_down(x, y)
181                 solution[(x, y + w)] = x, y
182                 y = y + w
183                 visited.append((x, y))
```

```
                visited.append((x, y))
                stack.append((x, y))

            elif cell_chosen == "down":
                adjacencey_mat[p][s] = 1
                adjacencey_mat[s][p] = 1

                push_down(x, y)
                solution[(x, y + w)] = x, y
                y = y + w
                visited.append((x, y))
                stack.append((x, y))

            elif cell_chosen == "up":
                adjacencey_mat[p][t] = 1
                adjacencey_mat[t][p] = 1

                push_up(x, y)
                solution[(x, y - w)] = x, y
                y = y - w
                visited.append((x, y))
                stack.append((x, y))
        else:
            x, y = stack.pop()                          # if no cells are available pop one from the stack
            single_cell(x, y)                           # use single_cell function to show backtracking image
            time.sleep(.03)                             # slow program down a bit
            backtracking_cell(x, y)                     # change colour to green to identify backtracking path
```

To support the carve_out_maze function we require to create the following sub-functions that help it. Refer the code comments for better understanding.

```

def push_up(x, y):
    pygame.draw.rect(screen, BLUE, (x + 1, y - w + 1, 19, 39), 0)       # draw a rectangle twice the width of the cell
    pygame.display.update()                                             # to animate the wall being removed


def push_down(x, y):
    pygame.draw.rect(screen, BLUE, (x + 1, y + 1, 19, 39), 0)
    pygame.display.update()


def push_left(x, y):
    pygame.draw.rect(screen, BLUE, (x - w + 1, y + 1, 39, 19), 0)
    pygame.display.update()


def push_right(x, y):
    pygame.draw.rect(screen, BLUE, (x + 1, y + 1, 39, 19), 0)
    pygame.display.update()

```

```
def single_cell(x, y):
    pygame.draw.rect(screen, YELLOW, (x + 1, y + 1, 18, 18), 0)         # draw a single width cell
    pygame.display.update()


def backtracking_cell(x, y):
    pygame.draw.rect(screen, BLUE, (x + 1, y + 1, 18, 18), 0)           # used to re-colour the path after single_cell
    pygame.display.update()                                             # has visited cell


def solution_cell(x, y):
    pygame.draw.rect(screen, YELLOW, (x + 8, y + 8, 5, 5), 0)           # used to show the solution
    pygame.display.update()                                             # has visited cell


def solution_cell_2(x, y):
    pygame.draw.rect(screen, GREEN, (x + 8, y + 8, 5, 5), 0)            # used to show the solution
    pygame.display.update()                                             # has visited cell
```

```
def pixel_to_vertex(x, y):
    row = (y / 20) - 1
    column = (x / 20) - 1
    vertex = row * row_col_size + column
    vertex = int(vertex)
    return vertex

def vertex_to_pixel(x):
    row = x / row_col_size
    row = int(row)
    column = x % row_col_size
    a = column * row_col_size + 20
    b = row * row_col_size + 20
    a = int(a)
    b = int(b)
    return a, b
```

4. Solving the maze

Using the Recursive Backtracking

As mentioned earlier, the “solution” HashMap already contains the respective parent of each and every cell, which was created while carving the maze. So, to find the shortest possible path from cell to cell 399, we will ask cell 399 to go its parent cell and then further asking for their parent cells till we reach cell 0 (whose parent is -1).

```
253
254     def plot_route_back(x, y):
255         solution_cell(x, y)                    # solution list contains all the coordinates to route back to start
256         while (x, y) != (20, 20):              # loop until cell position == start position
257             x, y = solution[x, y]              # "key value" now becomes the new key
258             solution_cell(x, y)                # animate route back
259             time.sleep(.03)
260
```

So, after running this function we will have our shortest path plotted in yellow square shaped signs as shown in figure 2.

Using the Dijkstra’s Algorithm

Once, the graph is represented in the form of adjacency matrix, it is not that difficult to implement the Dijkstra’s algorithm to find the shortest path. Since, we have covered this algorithm in our academics too. This function will return the list starting from the destination cell to the source cell, which gives the shortest possible path.

```
def dijkstra(src, dest):
    def min_distance(dist, sp_set):                    # choose b/n a vertex from set of vertices connected to parent
        min = 10 ** 10
        global min_index
        for v in range(400):                           # minimum distant adjacent vertex is chosen
            if sp_set[v] == False and dist[v] <= min:
                min = dist[v]
                min_index = v
        return min_index

    graph = copy.deepcopy(adjacencey_mat)
    parent = [-2 for i in range(400)]                  # every vertex keep track of its parent vertex

    dist = [10 ** 10 for i in range(size)]             # stores the dist w.r.t to src
    sp_set = [False for i in range(size)]              # tells whether already selected or covered along path
    dist[src] = 0
    parent[src] = -1

    for i in range(size - 1):
        u = min_distance(dist, sp_set)                 # returns the minimum distant adjacent vertex
        sp_set[u] = True
        # find all the vertices connected to the selected vertex u
        for v in range(size):
            if sp_set[v] is False and graph[u][v] != 0 and dist[u] != 10 ** 10 and dist[u] + graph[u][v] < dist[v]:
                dist[v] = dist[u] + graph[u][v]
                parent[v] = u
```

```
        # find all the vertices connected to the selected vertex u
        for v in range(size):
            if sp_set[v] is False and graph[u][v] != 0 and dist[u] != 10 ** 10 and dist[u] + graph[u][v] < dist[v]:
                dist[v] = dist[u] + graph[u][v]
                parent[v] = u

    def ancestor(des):
        list1 = []
        stop = des
        while parent[stop] != -1:                      # process of finding ancestory of destination vertex
            list1.append(parent[stop])
            stop = parent[stop]

        return list1

    destination_parent = ancestor(dest)
    print(destination_parent)

    for index in range(len(destination_parent)):
        e, f = vertex_to_pixel(dest)
        solution_cell_2(e, f)
        s = destination_parent[index]
        m, n = vertex_to_pixel(s)
        solution_cell_2(m, n)
        time.sleep(.03)
```

Hence, running this function will plot the shortest path from the given source cell to the destination cell as shown in figure 3.

5. References

- Wikipedia
- Python tutorials
- Pygame documentations
- tnypdavis code
- Dijkstra’s code help from ishaan sharma