Home Assigment 1, CMPE 252, Section 01, Spring 2023, San Jose State University

Informative Search using A Algorithm and its comparison to uninformed search methods (BFS, Dijkstra)*

All the required utility functions are provided at the beginning of this notebook. There are 8 tasks after the utility functions, and a bonus task (10 additional points to HW1, if solved correclty). **This assignment is individual**. The deadline is Feb 26, 2023, 11:59PM.The submission is in Canvas.

please submit two separate files (not in a ZIP file) this notebook and its corresponding PDF (File->Download as -> PDF)

import the necessary libraries

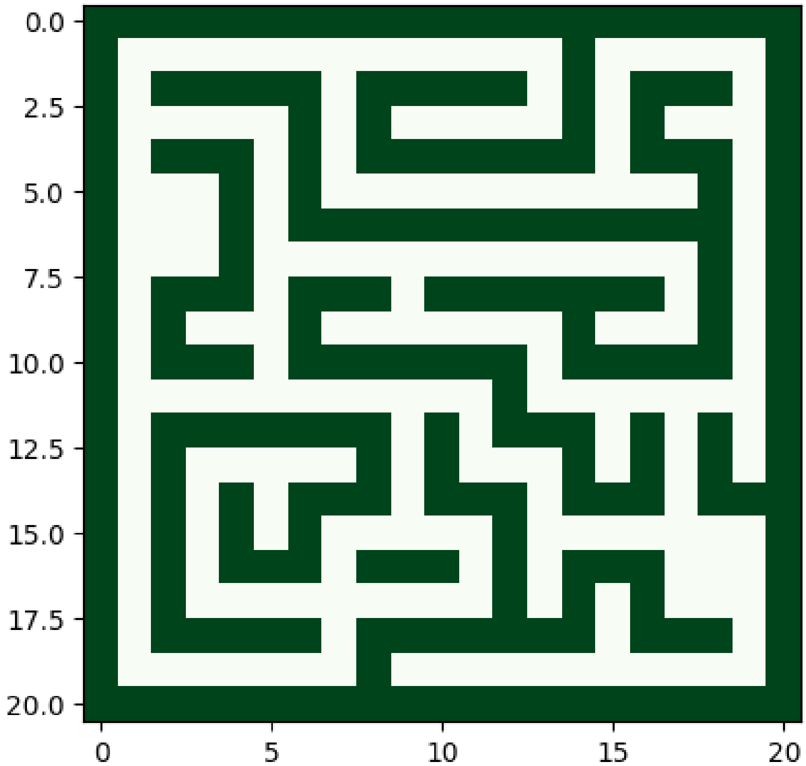
```
import matplotlib.pyplot as plt
import numpy as np
import sys
import networkx as nx
import time
%matplotlib inline
```

'build_maze' builds the maze from 'maze_file.txt'.

```
def build_maze(maze_file):
    '''
    para1: filename of the maze txt file
    return mazes as a numpy array walls: 0 - no wall, 1 - wall in the
maze
    '''
    a = open(maze_file, 'r')
    m=[]
    for i in a.readlines():
        m.append(np.array(i.split(" "), dtype="int32"))
    return np.array(m)
```

```
# (you are encouraged to look at the API of 'imshow')
plt.imshow(build_maze("maze_20x20.txt"), cmap='Greens')
```

```
<matplotlib.image.AxesImage at 0x1380ec670>
```



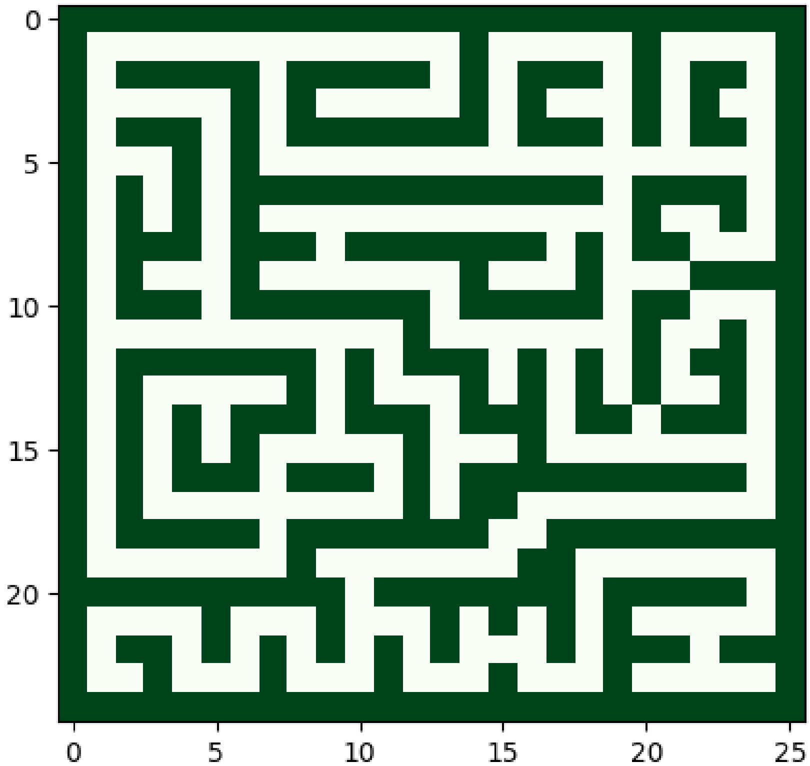
Task - 1

Build your maze with dimentions 25 x 25 and a similar complexity (number of obstacles/fences) as in the maze provided in 'maze_20x20.txt'. Check that there exists a path between START at the (1, 1) and the GOAL at (25, 25) in your maze. Store your maze to 'my_maze_25x25.txt'. Visualize your maze. Use your maze in the below tasks.

Visualize the maze:

```
# (you are encouraged to look at the API of 'imshow')
plt.imshow(build_maze("my_maze_25x25.txt"), cmap='Greens')
```

```
<matplotlib.image.AxesImage at 0x1381342e0>
```



define START and GOAL states within the maze

```
START=(1, 1)
GOAL=(23,24)
# Goal for 50X50 maze is (1,49)
```

'Find_the_edges' builds the graph for the maze, assuming that the robot can move only in the four directions (Up, Down, Right, Left).

```
def Find_the_edges(maze):
    '''
    para1: numpy array of the maze structure
    return graph of the connected nodes
    '''
    graph={}
    grid_size=len(maze)
    for i in range(len(maze)):
        for j in range(len(maze[0])):
            if(maze[i][j]!=1):
                adj=[]
                eles=[]
                if i - 1 >= 0:
                    eles.append((i-1,j))
                if i + 1 < grid_size:
                    eles.append((i+1,j))
                if j - 1 >=0:
```

```
                        eles.append((i,j-1))
                    if j+1< grid_size:
                        eles.append((i,j+1))
                    for ele in eles:
                        if maze[ele[0]][ele[1]] == 0 or maze[ele[0]]
[ele[1]]=='3' :
                            adj.append((ele[0],ele[1]))
                    graph[(i,j)]=adj
    return graph
```

- ***gray cells*** means the walls of the maze
- ***dark green cells*** means the visited cells of the maze
- ***light green cells*** means the shortest path of the maze
- ***light brown*** means the unvisited cells of the maze

Task - 2

A* algorithm requires a heuristic function. You will try two following heuristics:

- Euclidean distance between the cell coordinates
- Manhattan distance between the cell coordinates

```
# implement the Euclidean and Manhattan distance between the 2 nodes,
and update the code for A* accordingly

def Euclidean_distance(node1, node2):
    '''
    para1: is a tuple which contains the coorinates of the source node
    para2: is a tuple which contains the coorinates of the source node
    return: Euclidean distance between the 2 nodes
    '''
    return ((node1[0] - node2[0])**2 + (node1[1] - node2[1])**2)**0.5
    #pass

def Manhattan_distance(node1, node2):
    '''
    para1: is a tuple which contains the coorinates of the source node
    para2: is a tuple which contains the coorinates of the source node
    return: Manhattan distance between the 2 nodes
    '''
    #refer to https://xlinux.nist.gov/dads/HTML/manhattanDistance.html
    return abs(node1[0] - node2[0]) + abs(node1[1] - node2[1])
    #pass
```

Run A* with these two heuristic functions for W=1 and find the shortest path and its length in the maze. You can update the interface of astar_path to accept W and a heuristic function

A* -search

```
import heapq

class PriorityQueue:
    def __init__(self):
        self.elements = []

    def empty(self) -> bool:
        return not self.elements

    def put(self, item, priority):
        heapq.heappush(self.elements, (priority, item))

    def get(self):
        return heapq.heappop(self.elements)[1]

def astar_path(graph, start, goal, W, heuristic):
    '''
    para1: connected graph
    para2: Starting node
    para3: ending Node
    return1: list of visited nodes
    return2: nodes of shortest path
    '''
    frontier = PriorityQueue()
    frontier.put(start, 0)
    came_from= {}
    cost_so_far= {}
    came_from[start] = None
    cost_so_far[start] = 0
    while not frontier.empty():
        current = frontier.get()
        if current == goal:
            break
        #print(graph[current])
        for next in (graph[current]):
            maze1[current]=-1
            new_cost = cost_so_far[current] + 1
            if next not in cost_so_far or new_cost <
cost_so_far[next]:
                cost_so_far[next] = new_cost
                ######
                #you can make the interface of 'astar_path' more
robust by providing a heuristic as a parameter
                ###
                priority = new_cost + W * heuristic(next, goal)
                frontier.put(next, priority)
                came_from[next] = current
    current= goal
    path = []
    while current != start:
```

```
        path.append(current)
        #print(came_from[current])
        current = came_from[current]
    path.append(start)
    path.reverse()
    return came_from, path
'''
visited nodes - mark them as -3 in maze numpy array
path- mark them as -1 in maze numpy array
and Visualize the maze
'''
```

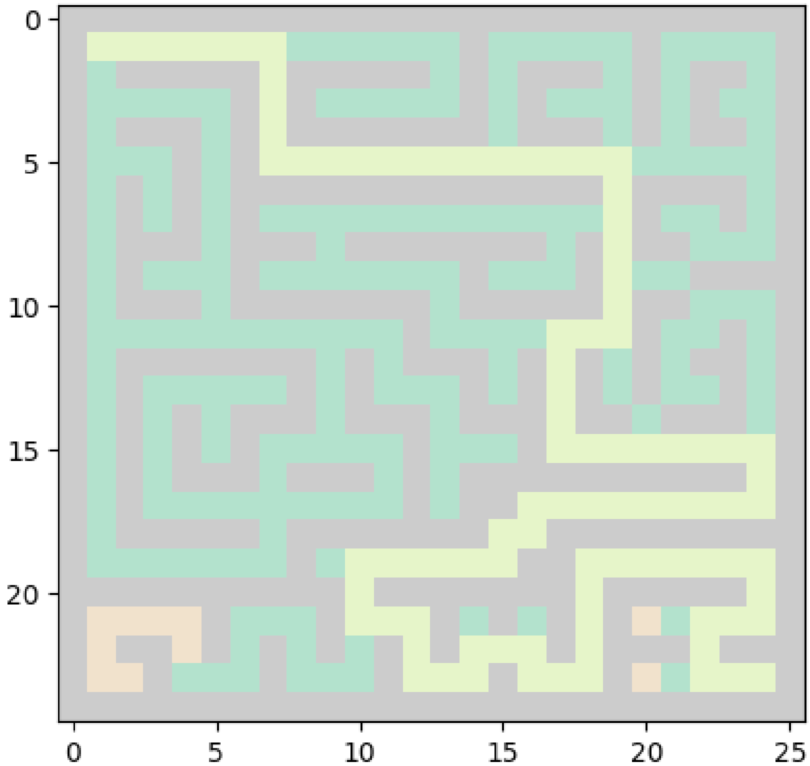
```
'\nvisited nodes - mark them as -3 in maze numpy array \npath- mark
them as -1 in maze numpy array\nand Visualize the maze\n'
```

```
weights_euclidean = []
weights_manhattan = []
time_taken_euclidean = []
time_taken_manhattan = []
visited_nodes_euclidean = []
visited_nodes_manhattan = []
```

A* search with Eucliedean distance as heuristic function

```
# A* search with Eucliedean distance as heuristic function
# visited nodes are marked by '-3', the final path is marked by '-1'.
maze1 = build_maze("my_maze_25x25.txt")
graph=Find_the_edges(maze1)
W = 1

start_time= time.time()
came_from, path = astar_path(graph, START, GOAL, W,
Euclidean_distance)

weights_euclidean.append(W)
time_taken_euclidean.append(time.time() - start_time)
visited_nodes_euclidean.append(len(came_from))


for i in came_from:
    maze1[i[0],i[1]]=-3
for i in path:
    maze1[i[0],i[1]]=-1

plt.imshow(maze1, cmap='Pastel2')
```

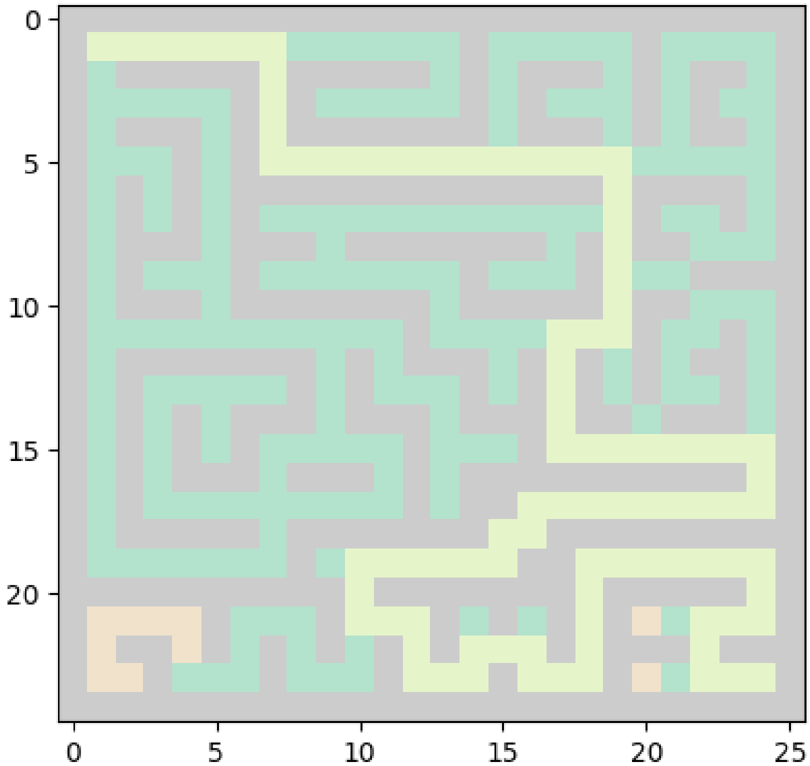
```
<matplotlib.image.AxesImage at 0x1397009a0>
```



A* search with Manhattan distance as heuristic function

```
# A* search with Manhattan distance as heuristic function
# visited nodes are marked by '-3', the final path is marked by '-1'.
maze1=build_maze("my_maze_25x25.txt")
graph=Find_the_edges(maze1)
W = 1

start_time= time.time()
came_from, path = astar_path(graph, START, GOAL, W,
Manhattan_distance)

weights_manhattan.append(W)
time_taken_manhattan.append(time.time() - start_time)
visited_nodes_manhattan.append(len(came_from))


for i in came_from:
    maze1[i[0],i[1]]=-3
for i in path:
    maze1[i[0],i[1]]=-1

plt.imshow(maze1, cmap='Pastel2')
```

```
<matplotlib.image.AxesImage at 0x13b17d760>
```



Task - 3

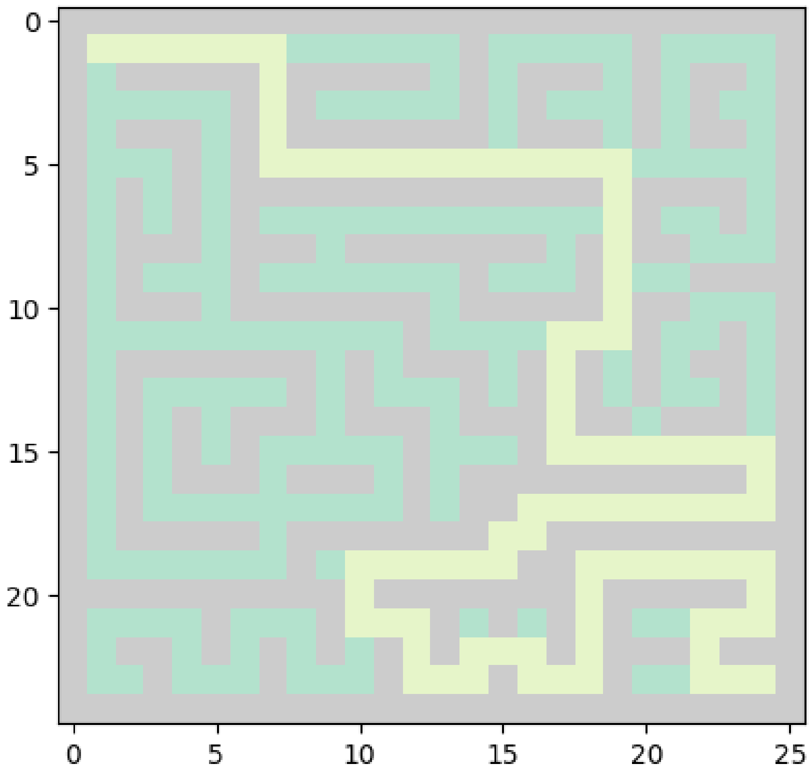
In this task you are asked to solve the maze with 4 different weights, W, in A* for each of the heurstic function mentioned above. Visualize the solution for each W and each heurstic on a separate plot in the same format as in the examle above (see cell 17). ***Chose a broad set of values for W to see the difference.***

A* Search | W => 0.1 | Heuristic => Euclidean_distance

```
maze1=build_maze("my_maze_25x25.txt")
graph=Find_the_edges(maze1)
W = 0.1

start_time= time.time()
came_from, path = astar_path(graph, START, GOAL, W,
Euclidean_distance)

weights_euclidean.append(W)
time_taken_euclidean.append(time.time() - start_time)
visited_nodes_euclidean.append(len(came_from))

for i in came_from:
    maze1[i[0],i[1]]=-3
for i in path:
```

```
    maze1[i[0],i[1]]=-1

plt.imshow(maze1, cmap='Pastel2')
```

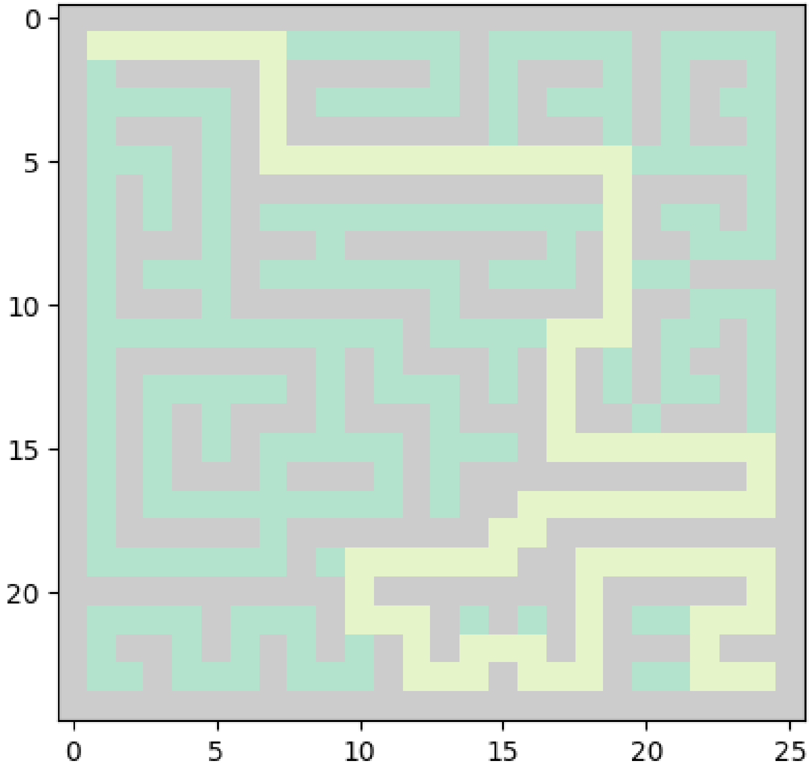
```
<matplotlib.image.AxesImage at 0x13b73f040>
```



A* Search | W => 0.3 | Heuristic => Euclidean_distance

```
maze1=build_maze("my_maze_25x25.txt")
graph=Find_the_edges(maze1)
W = 0.3
start_time= time.time()
came_from, path = astar_path(graph, START, GOAL, W,
Euclidean_distance)

weights_euclidean.append(W)
time_taken_euclidean.append(time.time() - start_time)
visited_nodes_euclidean.append(len(came_from))

for i in came_from:
    maze1[i[0],i[1]]=-3
for i in path:
    maze1[i[0],i[1]]=-1

plt.imshow(maze1, cmap='Pastel2')
```

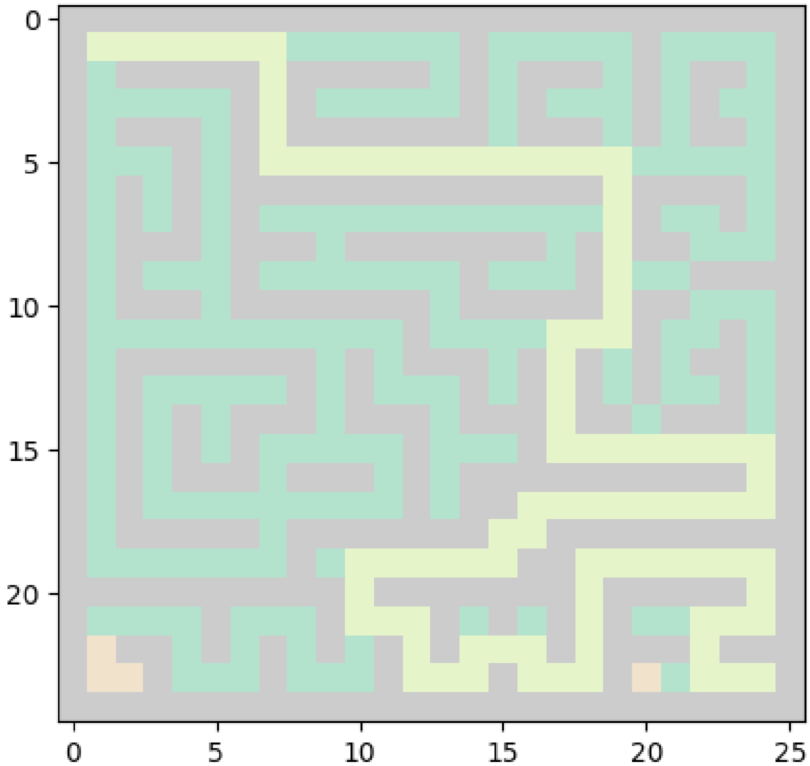
```
<matplotlib.image.AxesImage at 0x13b9098b0>
```



A* Search | W => 0.6 | Heuristic => Euclidean_distance

```
maze1=build_maze("my_maze_25x25.txt")
graph=Find_the_edges(maze1)
W = 0.6
start_time= time.time()
came_from, path = astar_path(graph, START, GOAL, W,
Euclidean_distance)

weights_euclidean.append(W)
time_taken_euclidean.append(time.time() - start_time)
visited_nodes_euclidean.append(len(came_from))

for i in came_from:
    maze1[i[0],i[1]]=-3
for i in path:
    maze1[i[0],i[1]]=-1

plt.imshow(maze1, cmap='Pastel2')
```

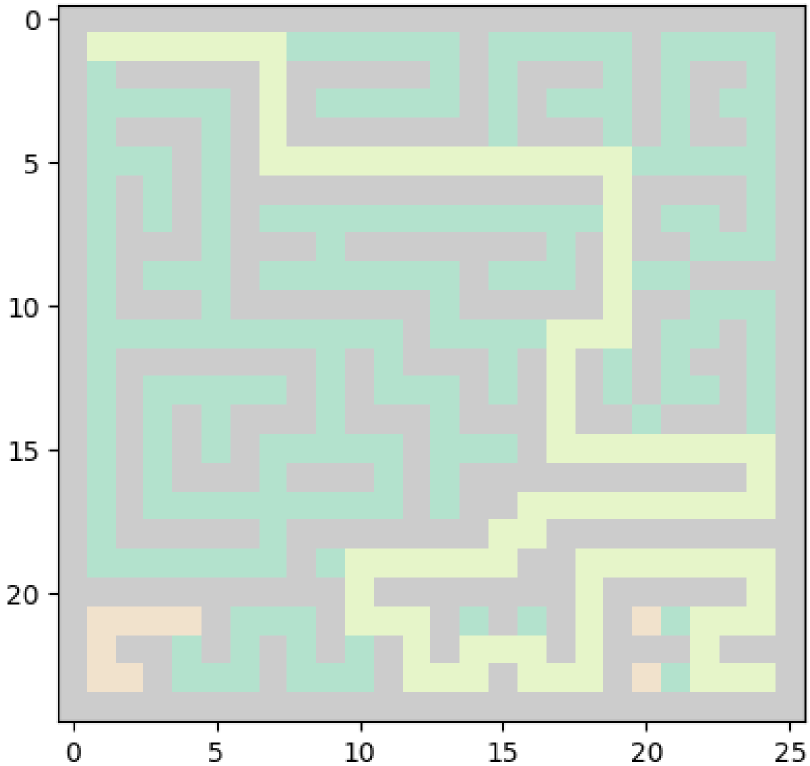
```
<matplotlib.image.AxesImage at 0x13b96ca90>
```



A* Search | W => 0.9 | Heuristic => Euclidean_distance

```
maze1=build_maze("my_maze_25x25.txt")
graph=Find_the_edges(maze1)
W = 0.9
start_time= time.time()
came_from, path = astar_path(graph, START, GOAL, W,
Euclidean_distance)

weights_euclidean.append(W)
time_taken_euclidean.append(time.time() - start_time)
visited_nodes_euclidean.append(len(came_from))

for i in came_from:
    maze1[i[0],i[1]]=-3
for i in path:
    maze1[i[0],i[1]]=-1

plt.imshow(maze1, cmap='Pastel2')
```

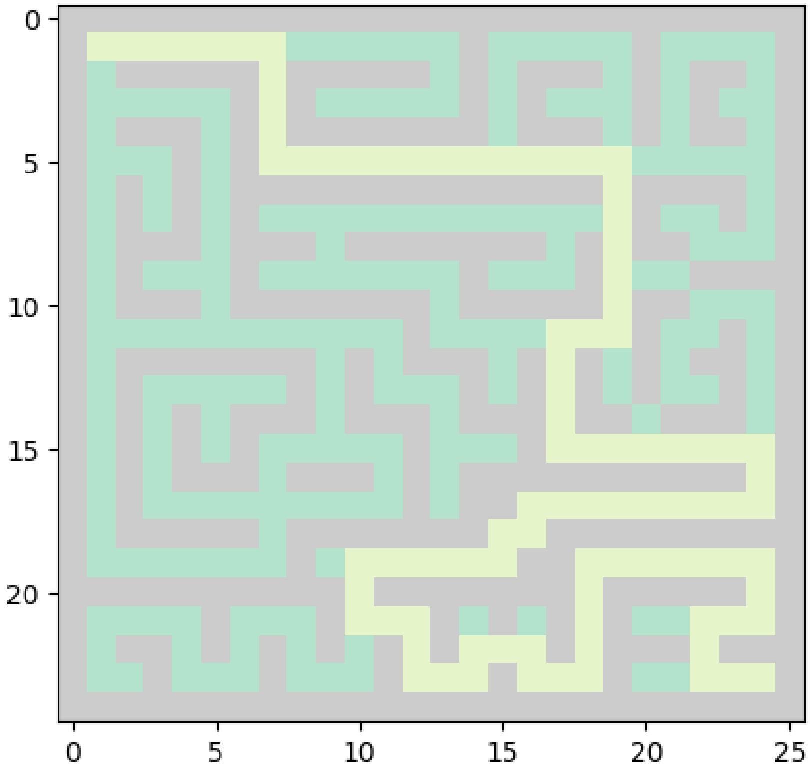
```
<matplotlib.image.AxesImage at 0x13b9ccb20>
```



A* Search | W => 0.1 | Heuristic => Manhattan_distance

```
maze1=build_maze("my_maze_25x25.txt")
graph=Find_the_edges(maze1)
W = 0.1
start_time= time.time()
came_from, path = astar_path(graph, START, GOAL, W,
Manhattan_distance)

weights_manhattan.append(W)
time_taken_manhattan.append(time.time() - start_time)
visited_nodes_manhattan.append(len(came_from))

for i in came_from:
    maze1[i[0],i[1]]=-3
for i in path:
    maze1[i[0],i[1]]=-1

plt.imshow(maze1, cmap='Pastel2')
```

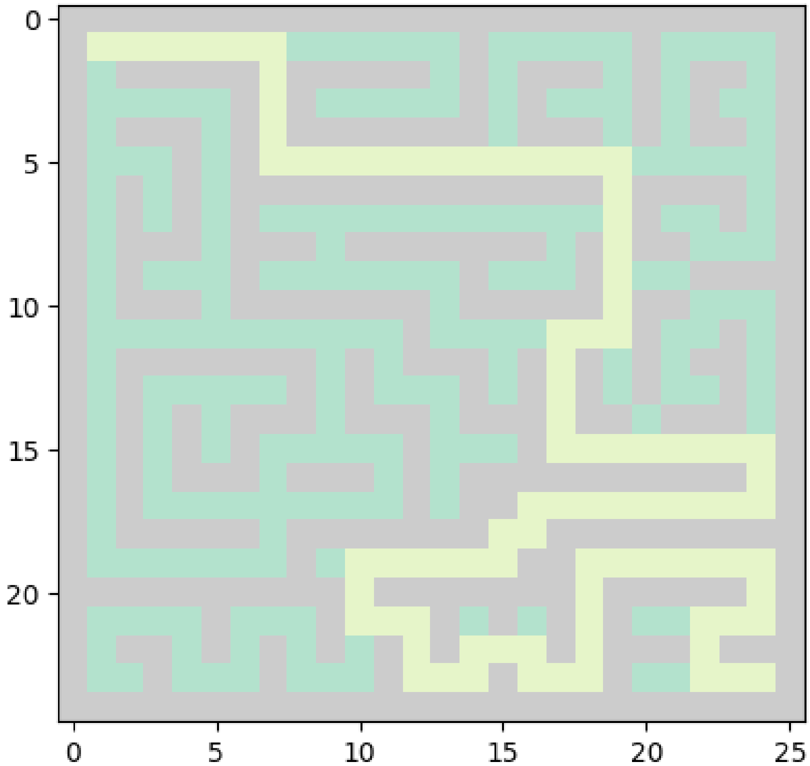
```
<matplotlib.image.AxesImage at 0x13ba2aa90>
```



A* Search | W => 0.3 | Heuristic => Manhattan_distance

```
maze1=build_maze("my_maze_25x25.txt")
graph=Find_the_edges(maze1)
W = 0.3
start_time= time.time()
came_from, path = astar_path(graph, START, GOAL, W,
Manhattan_distance)

weights_manhattan.append(W)
time_taken_manhattan.append(time.time() - start_time)
visited_nodes_manhattan.append(len(came_from))

for i in came_from:
    maze1[i[0],i[1]]=-3
for i in path:
    maze1[i[0],i[1]]=-1

plt.imshow(maze1, cmap='Pastel2')
```

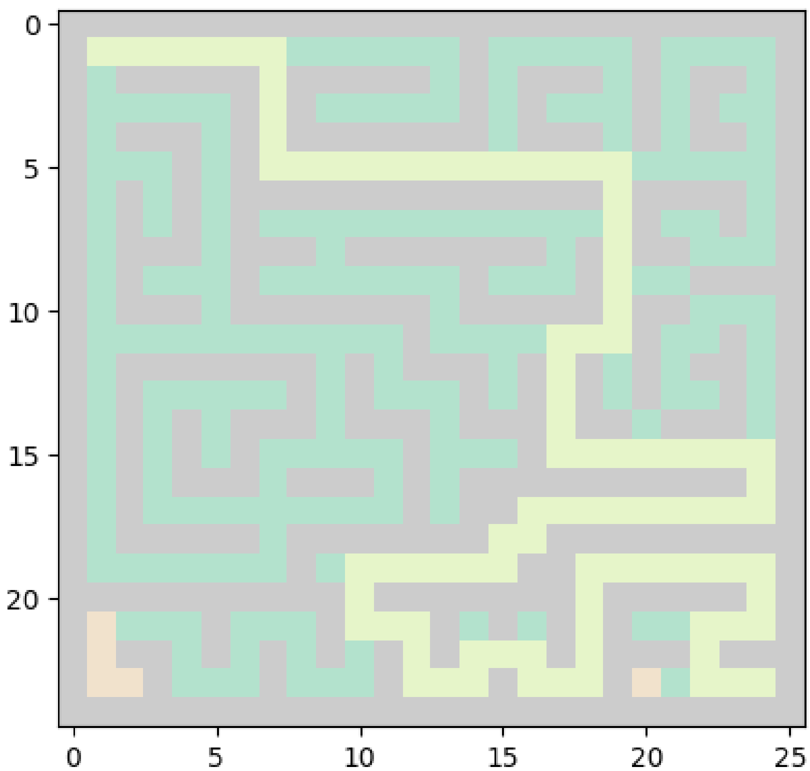
```
<matplotlib.image.AxesImage at 0x13ba8bca0>
```



A* Search | W => 0.6 | Heuristic => Manhattan_distance

```
maze1=build_maze("my_maze_25x25.txt")
graph=Find_the_edges(maze1)
W = 0.6
start_time= time.time()
came_from, path = astar_path(graph, START, GOAL, W,
Manhattan_distance)

weights_manhattan.append(W)
time_taken_manhattan.append(time.time() - start_time)
visited_nodes_manhattan.append(len(came_from))

for i in came_from:
    maze1[i[0],i[1]]=-3
for i in path:
    maze1[i[0],i[1]]=-1

plt.imshow(maze1, cmap='Pastel2')
```

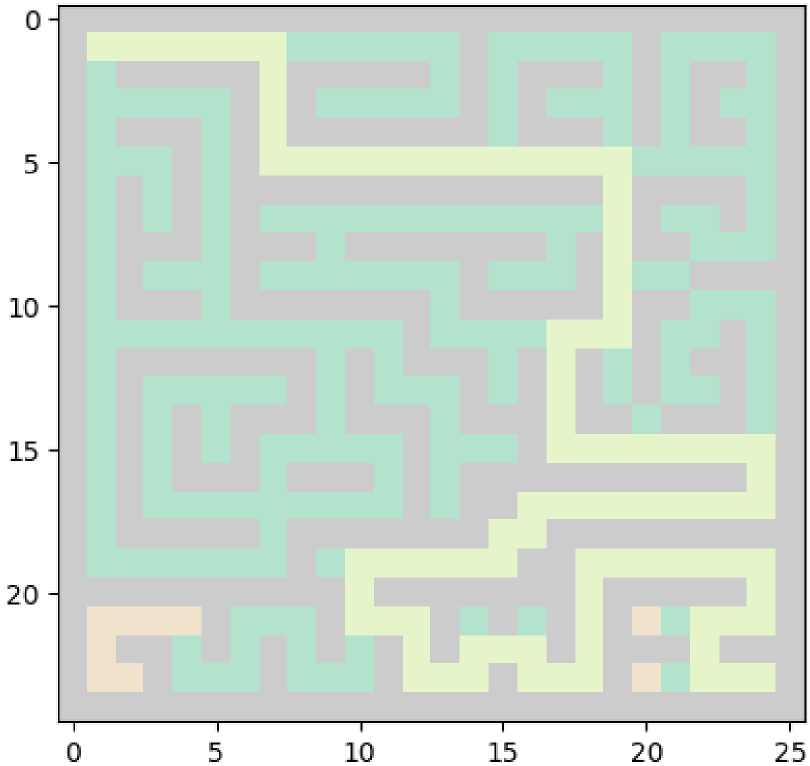
```
<matplotlib.image.AxesImage at 0x13baeab50>
```



A* Search | W => 0.9 | Heuristic => Manhattan_distance

```
maze1=build_maze("my_maze_25x25.txt")
graph=Find_the_edges(maze1)
W = 0.9
start_time= time.time()
came_from, path = astar_path(graph, START, GOAL, W,
Manhattan_distance)

weights_manhattan.append(W)
time_taken_manhattan.append(time.time() - start_time)
visited_nodes_manhattan.append(len(came_from))

for i in came_from:
    maze1[i[0],i[1]]=-3
for i in path:
    maze1[i[0],i[1]]=-1

plt.imshow(maze1, cmap='Pastel2')
```

```
<matplotlib.image.AxesImage at 0x13bb4bd90>
```



Explain what changes you observe for the different weights and why it occurs.

It is observed that as we increase weight, the number of states explored decrease for each heuristic function (Euclidean and Manhattan).

Task - 4

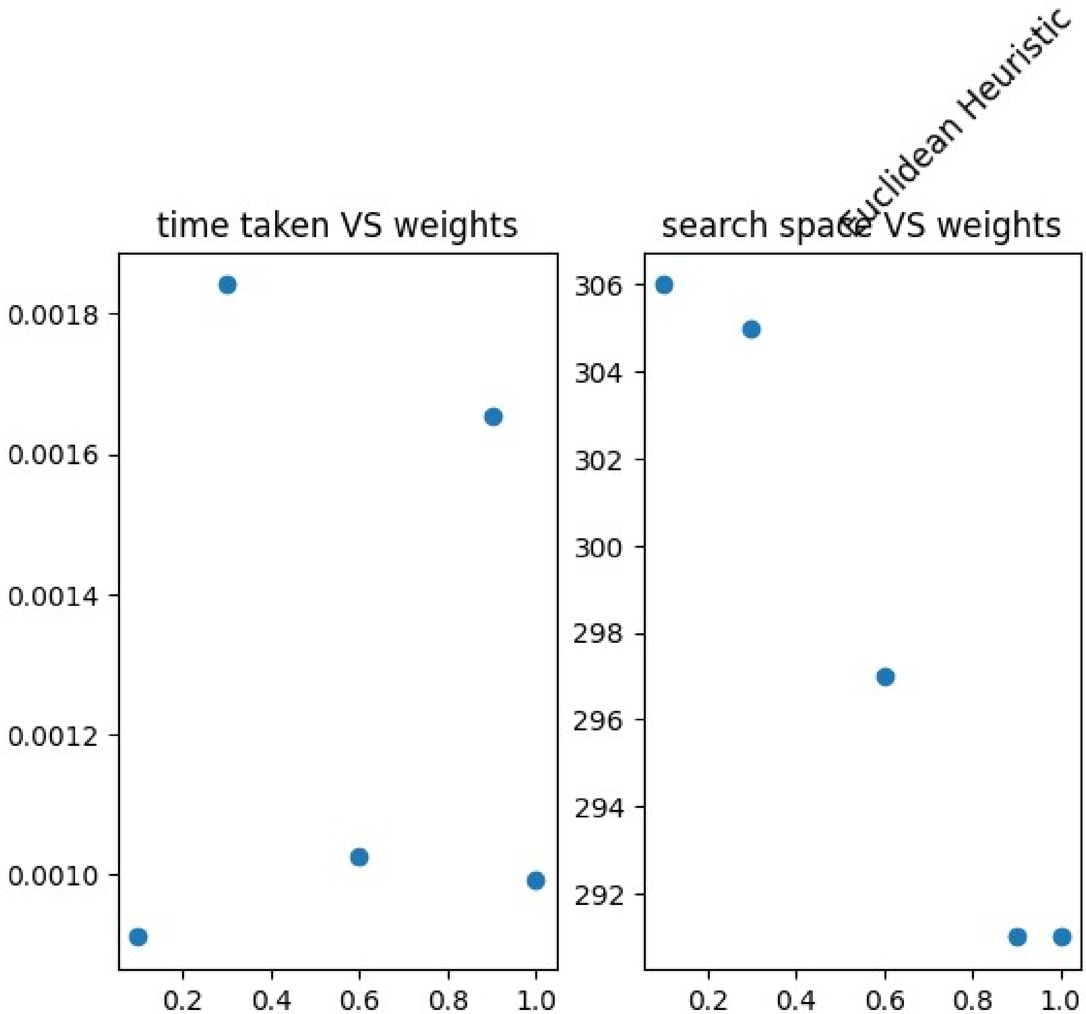
Plot on plt.subplot(121) a) time taken VS Weights

Plot on plt.subplot(122) b) search space (expanded nodes) VS Weights

-- add titles, axis labels, and legends.

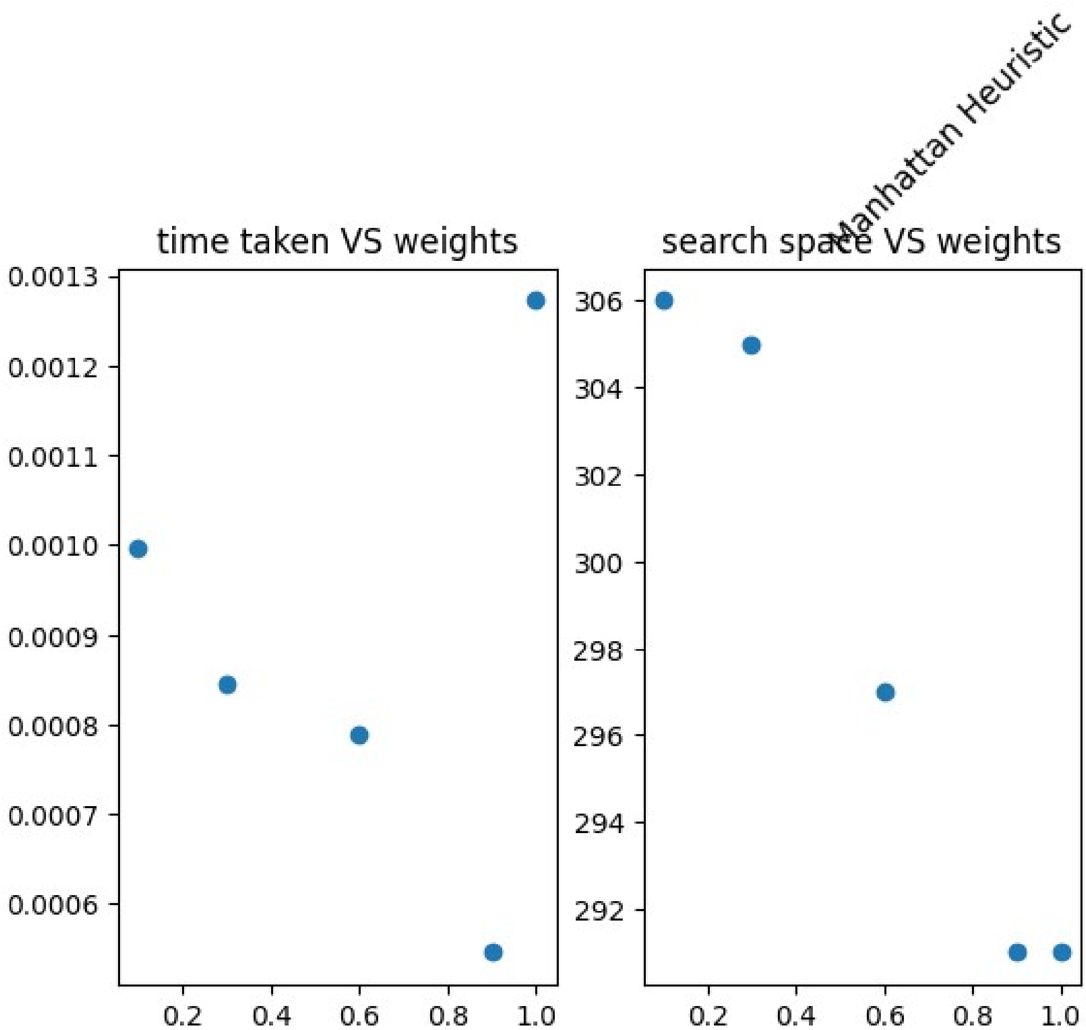
For Euclidean Heuristic function

```
fig = plt.figure()
fig.add_subplot(121, title = "time taken VS weights")
plt.scatter(weights_euclidean, time_taken_euclidean)
fig.add_subplot(122, title = "search space VS weights")
plt.scatter(weights_euclidean, visited_nodes_euclidean)
plt.title(label="Euclidean Heuristic", loc="right", rotation = 45)
plt.show()
```



For Manhattan Heuristic function

```
fig = plt.figure()
fig.add_subplot(121, title = "time taken VS weights")
plt.scatter(weights_manhattan, time_taken_manhattan)
fig.add_subplot(122, title = "search space VS weights")
plt.scatter(weights_manhattan, visited_nodes_manhattan)
plt.title(label="Manhattan Heuristic", loc="right", rotation = 45)
plt.show()
```



Task - 5

Solve the maze with the Dijkstra algorithm, and visualize the solution in the maze. What is the length of the shortest path?

Dijkstra Algorithm

```
def dijkstra_algorithm(graph, start_node, GOAL):
    '''
    para1: connected graph
    para2: Starting node
    para3: ending Node
    return1: list of visited nodes
    return2: nodes of shortest path
    '''
    unvisited_nodes = list(graph.keys())
```

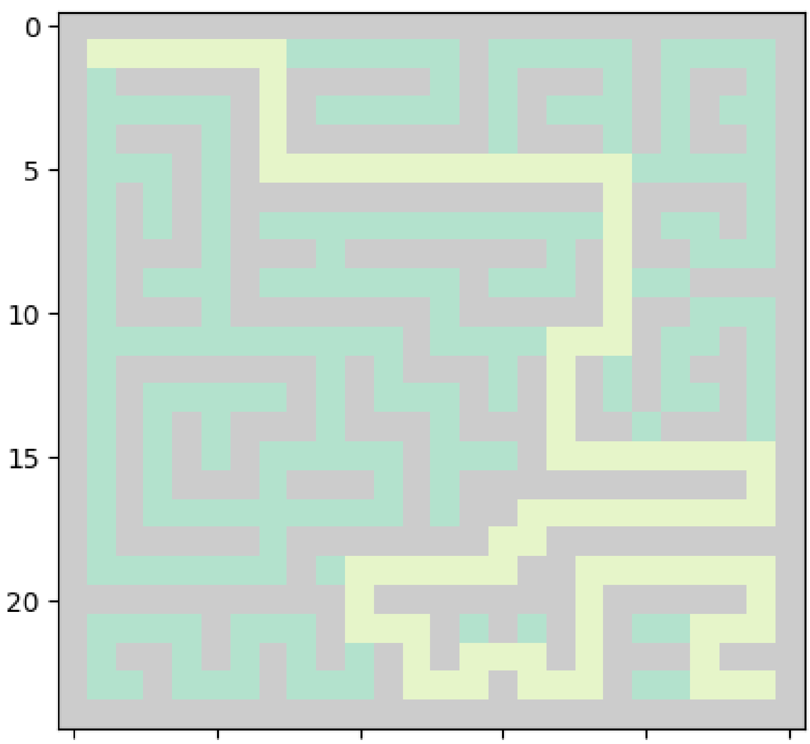
```
    # We'll use this dict to save the cost of visiting each node and
update it as we move along the graph
    shortest_path = {}

    # We'll use this dict to save the shortest known path to a node
found so far
    previous_nodes = {}

    # We'll use max_value to initialize the "infinity" value of the
unvisited nodes
    max_value = sys.maxsize
    for node in unvisited_nodes:
        shortest_path[node] = max_value
    # However, we initialize the starting node's value with 0
    shortest_path[start_node] = 0

    # The algorithm executes until we visit all nodes
    while GOAL in unvisited_nodes:
        # The code block below finds the node with the lowest score
        current_min_node = None
        for node in unvisited_nodes: # Iterate over the nodes
            if current_min_node == None:
                current_min_node = node
            elif shortest_path[node] <
shortest_path[current_min_node]:
                current_min_node = node

        # The code block below retrieves the current node's neighbors
and updates their distances
        neighbors = graph[current_min_node]
        for neighbor in neighbors:
            tentative_value = shortest_path[current_min_node] + 1
            if tentative_value < shortest_path[neighbor]:
                shortest_path[neighbor] = tentative_value
                # We also update the best path to the current node
                previous_nodes[neighbor] = current_min_node

        # After visiting its neighbors, we mark the node as "visited"
        unvisited_nodes.remove(current_min_node)

    current= GOAL
    path = []
    while current != start_node:
        path.append(current)
        #print(previous_nodes[current])
        current = previous_nodes[current]
    path.append(start_node)
    path.reverse()
    return previous_nodes, path
```

```python
'''
visited nodes - mark them as -3 in maze numpy array
path- mark them as -1 in maze numpy array
and Visualize the maze
'''
```

```
'\nvisited nodes - mark them as -3 in maze numpy array \npath- mark them as -1 in maze numpy array\nand Visualize the maze\n'
```

```python
maze1=build_maze("my_maze_25x25.txt")
graph=Find_the_edges(maze1)
previous_nodes, path = dijkstra_algorithm(graph, START, GOAL)
for i in previous_nodes:
    maze1[i[0],i[1]]=-3
for i in path:
    maze1[i[0],i[1]]=-1

plt.imshow(maze1, cmap='Pastel2')
```

```
<matplotlib.image.AxesImage at 0x13851e070>
```




```python
print("Length of shortest path via Dijkstras : ",len(path))
```

```
Length of shortest path via Dijkstras :  92
```

Task - 6

Solve the maze with the BFS algorithm, and visualize the solution in the maze. What is the length of the shortest path?

Breadth First Search (BFS)

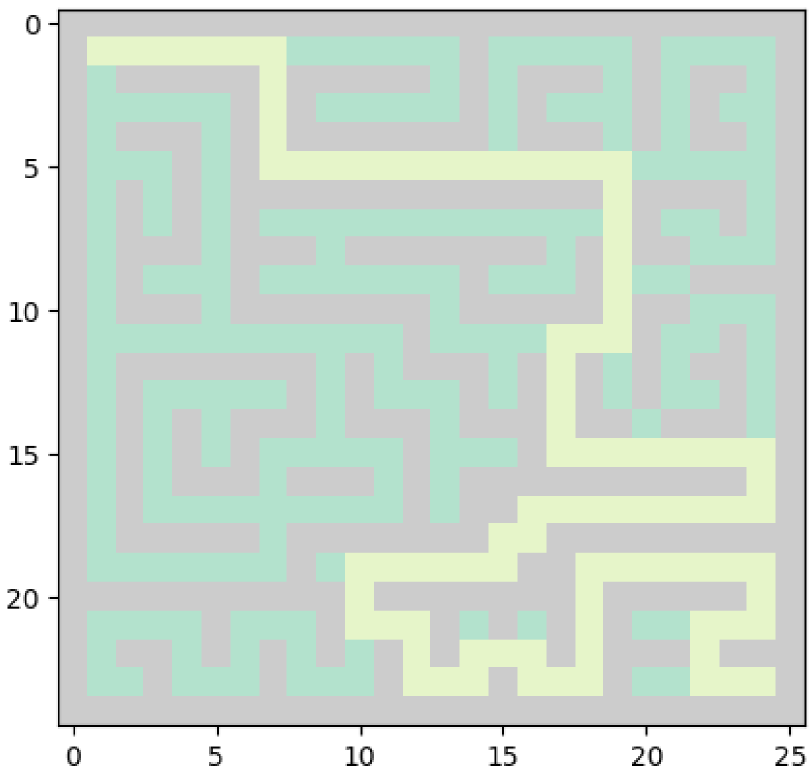
```
from collections import deque
def BreadthFirst(graph, start, goal):
    '''
    para1: connected graph
    para2: Starting node
    para3: ending Node
    return1: list of visited nodes
    return2: nodes of shortest path
    '''
    queue = deque([([start], start)])
    visited = set()

    while queue:
        path, current = queue.popleft()
        #print(path, current)
        if current == goal:
            #print( path)
            return visited, np.array(path)
        if current in visited:
            continue
        #print(current)
        visited.add(current)
        #print(current, graph[current])
        for neighbour in graph[current]:
            #print(graph[current])
            p = list(path)
            p.append(neighbour)
            queue.append((p, neighbour))
    return None
'''
visited nodes - mark them as -3 in maze numpy array
path- mark them as -1 in maze numpy array
and Visualize the maze
'''
```

```
'\nvisited nodes - mark them as -3 in maze numpy array \npath- mark
them as -1 in maze numpy array\nand Visualize the maze\n'
```

```
#example for visualization of maze with visited nodes and shortest
path
# visited nodes are marked by '-3', the final path is marked by '-1'.
maze1=build_maze("my_maze_25x25.txt")
graph=Find_the_edges(maze1)
#print(graph)
```

```
visited, path = BreadthFirst(graph, START, GOAL)
#print(visited, path)
for i in visited:
    maze1[i[0],i[1]]=-3
for i in path:
    maze1[i[0],i[1]]=-1

plt.imshow(maze1, cmap='Pastel2')
```

```
<matplotlib.image.AxesImage at 0x138579ee0>
```




```
print("Length of shortest path via BFS : ",len(path))
```

```
Length of shortest path via BFS :  92
```

Task - 7

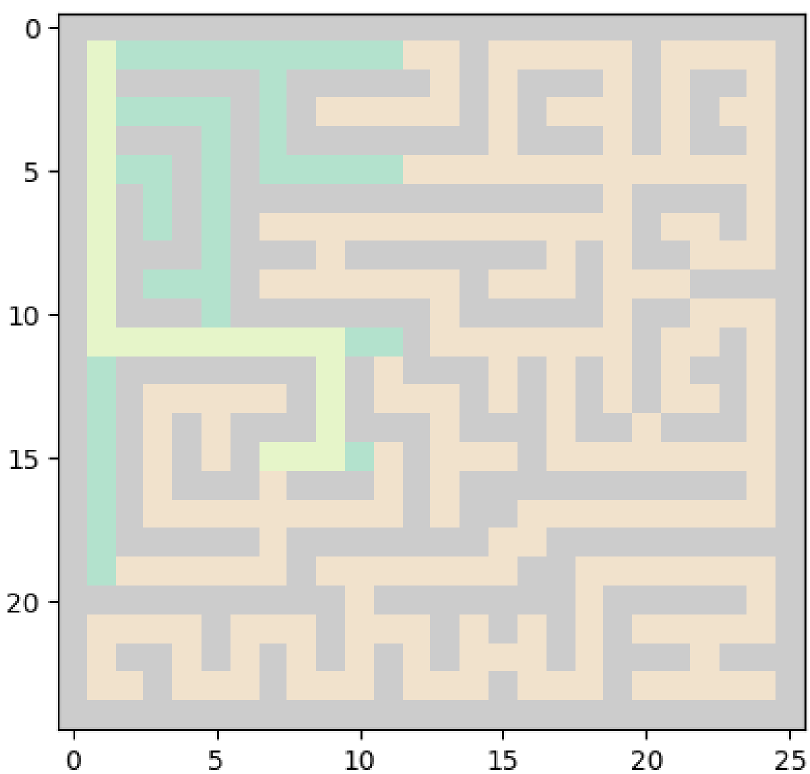
Choose 3 random START and GOAL states, and repeat the tasks 2 - 6, and visualize the solution for each. Use W=1 in this task. Explain your observations.

```
START1, GOAL1 = (1, 1), (15, 7)
START2, GOAL2 = (15, 7), (23, 24)
START3, GOAL3 = (1, 15), (1, 23)
```

A* search with Euclidean distance as heuristic function

```python
# A* search with Eucliedean distance as heuristic function
# visited nodes are marked by '-3', the final path is marked by '-1'.
maze1=build_maze("my_maze_25x25.txt")
graph=Find_the_edges(maze1)
W = 1
came_from, path = astar_path(graph, START1, GOAL1, W,
Euclidean_distance)
for i in came_from:
    maze1[i[0],i[1]]=-3

for i in path:
    maze1[i[0],i[1]]=-1

plt.imshow(maze1, cmap='Pastel2')
```

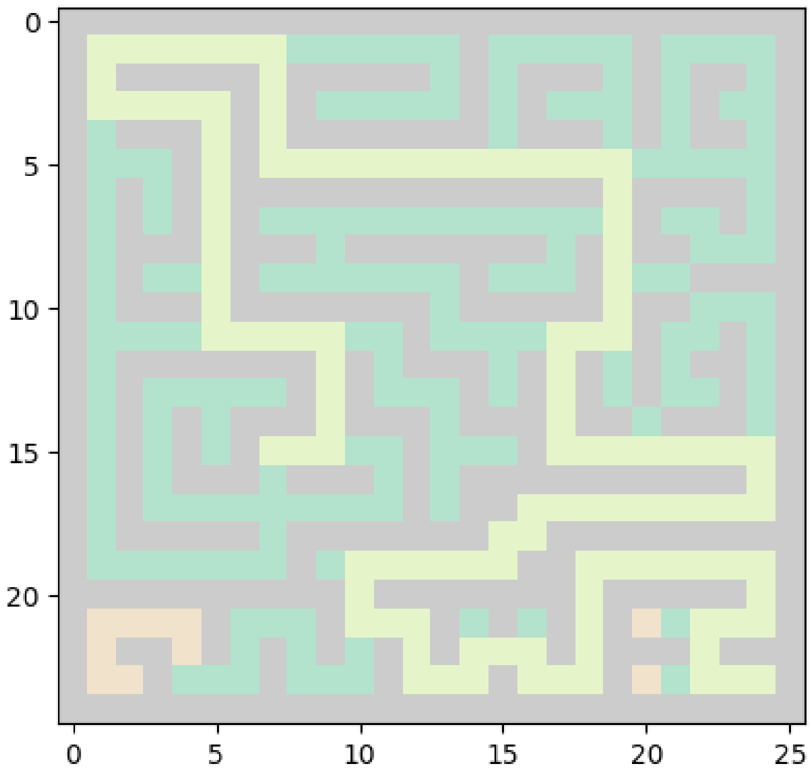
<matplotlib.image.AxesImage at 0x1385e6730>




```python
# A* search with Eucliedean distance as heuristic function
# visited nodes are marked by '-3', the final path is marked by '-1'.
maze1=build_maze("my_maze_25x25.txt")
graph=Find_the_edges(maze1)
W = 1
came_from, path = astar_path(graph, START2, GOAL2, W,
Euclidean_distance)
for i in came_from:
    maze1[i[0],i[1]]=-3
```

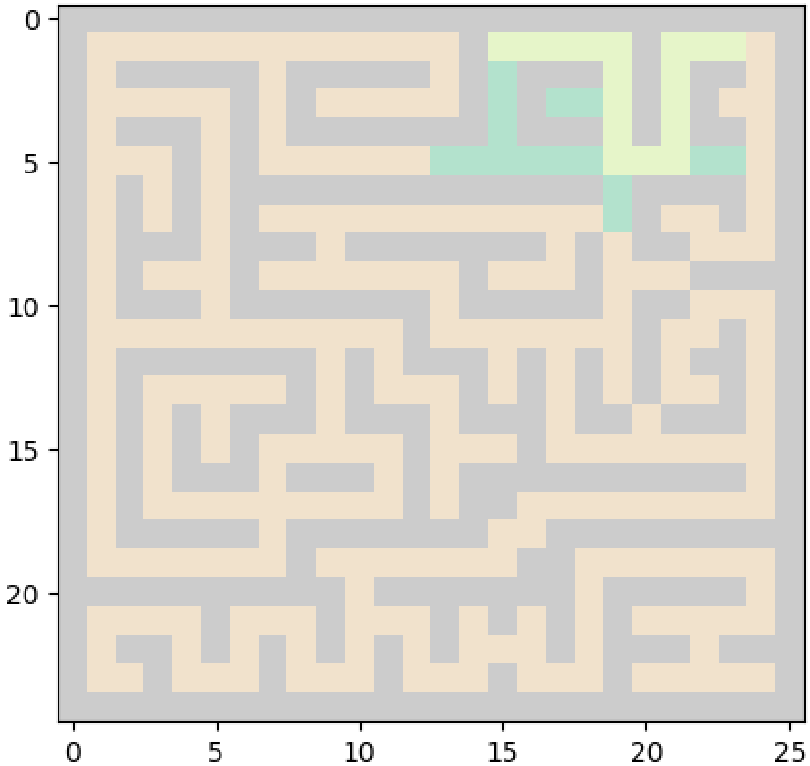
```python
for i in path:
    maze1[i[0],i[1]]=-1

plt.imshow(maze1, cmap='Pastel2')
```

```
<matplotlib.image.AxesImage at 0x138741ee0>
```



```python
# A* search with Eucliedean distance as heuristic function
# visited nodes are marked by '-3', the final path is marked by '-1'.
maze1=build_maze("my_maze_25x25.txt")
graph=Find_the_edges(maze1)
W = 1
came_from, path = astar_path(graph, START3, GOAL3, W,
Euclidean_distance)
for i in came_from:
    maze1[i[0],i[1]]=-3
for i in path:
    maze1[i[0],i[1]]=-1

plt.imshow(maze1, cmap='Pastel2')
```

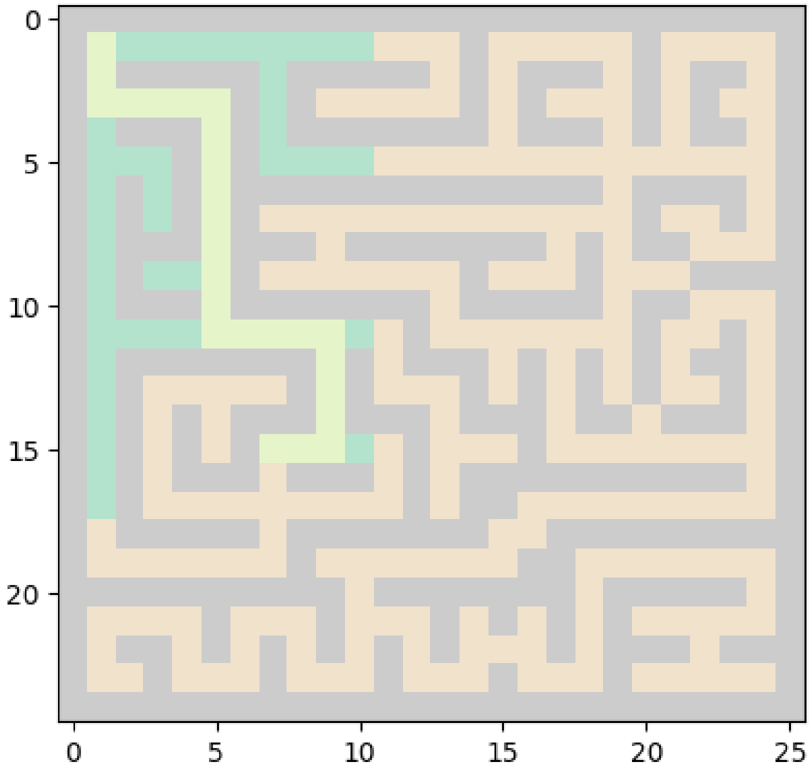
```
<matplotlib.image.AxesImage at 0x1387a1cd0>
```



A* search with Manhattan distance as heuristic function

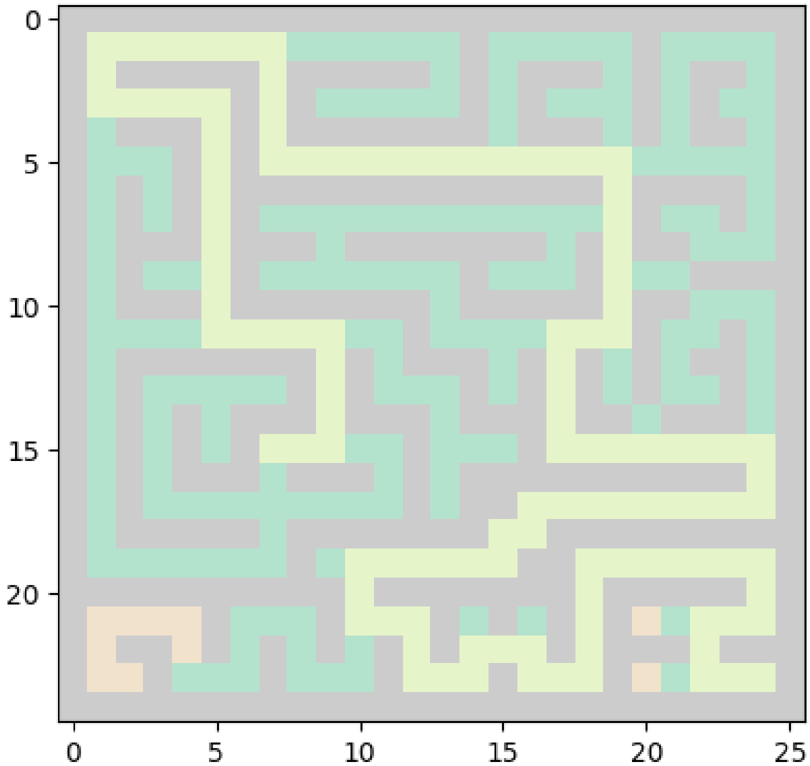
```
# A* search with Eucliedean distance as heuristic function
# visited nodes are marked by '-3', the final path is marked by '-1'.
maze1=build_maze("my_maze_25x25.txt")
graph=Find_the_edges(maze1)
W = 1
came_from, path = astar_path(graph, START1, GOAL1, W,
Manhattan_distance)
for i in came_from:
    maze1[i[0],i[1]]=-3

for i in path:
    maze1[i[0],i[1]]=-1

plt.imshow(maze1, cmap='Pastel2')
```

```
<matplotlib.image.AxesImage at 0x1390710a0>
```



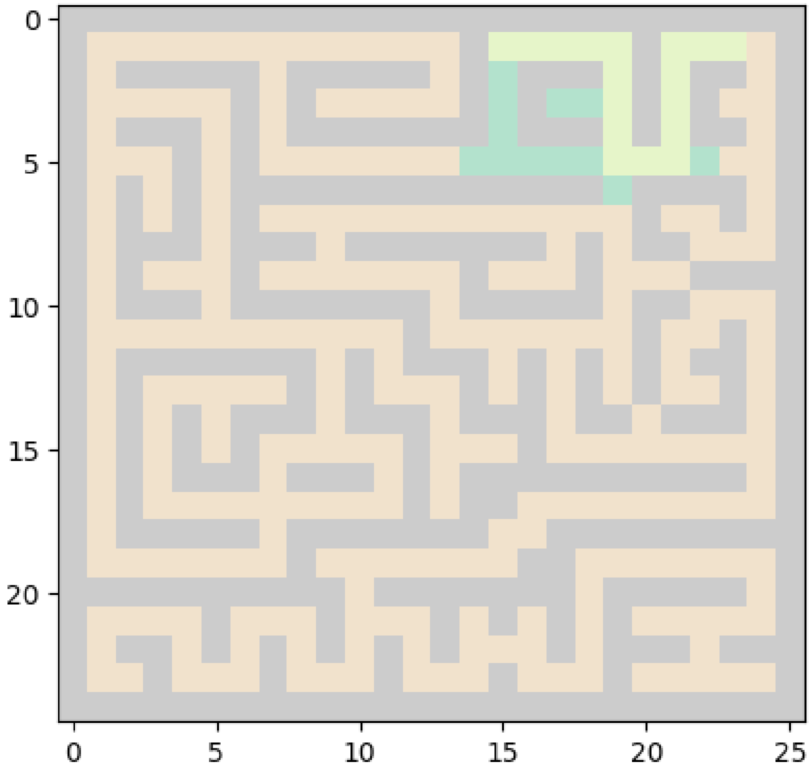

```
# A* search with Euclidean distance as heuristic function
# visited nodes are marked by '-3', the final path is marked by '-1'.
maze1=build_maze("my_maze_25x25.txt")
graph=Find_the_edges(maze1)
W = 1
came_from, path = astar_path(graph, START2, GOAL2, W,
Manhattan_distance)
for i in came_from:
    maze1[i[0],i[1]]=-3

for i in path:
    maze1[i[0],i[1]]=-1

plt.imshow(maze1, cmap='Pastel2')
```

```
<matplotlib.image.AxesImage at 0x1390c5820>
```




```
# A* search with Euclidean distance as heuristic function
# visited nodes are marked by '-3', the final path is marked by '-1'.
maze1=build_maze("my_maze_25x25.txt")
graph=Find_the_edges(maze1)
W = 1
came_from, path = astar_path(graph, START3, GOAL3, W,
Manhattan_distance)
for i in came_from:
    maze1[i[0],i[1]]=-3

for i in path:
    maze1[i[0],i[1]]=-1

plt.imshow(maze1, cmap='Pastel2')
```

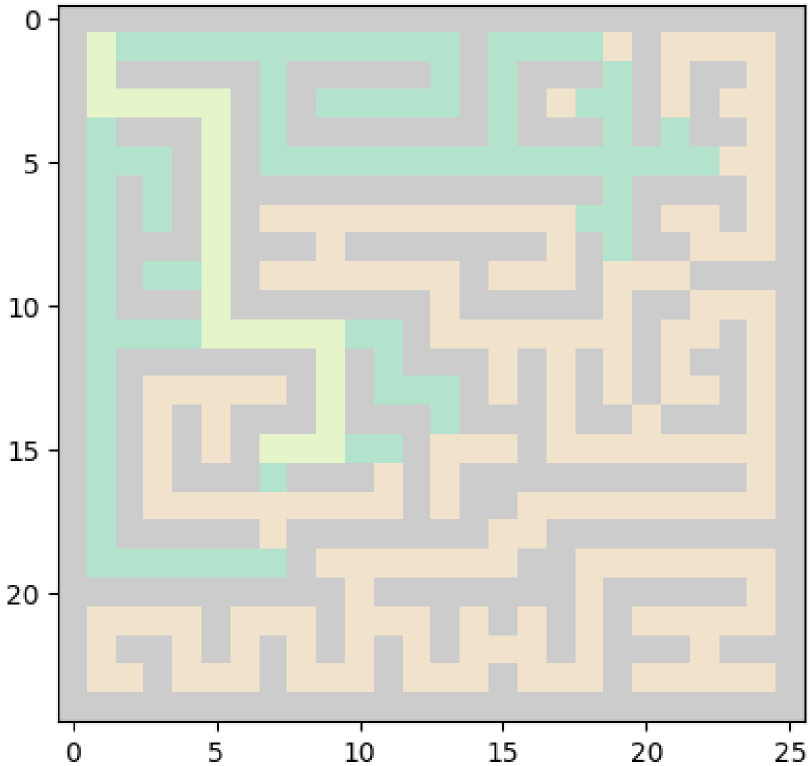
```
<matplotlib.image.AxesImage at 0x1391264c0>
```



Dijkstra's Algorithm

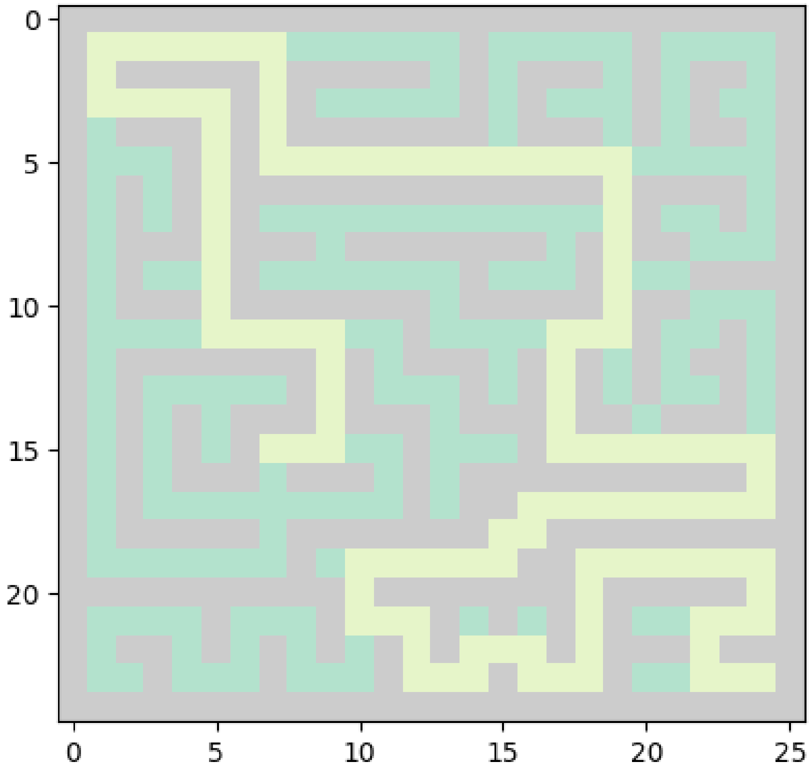
```
maze1=build_maze("my_maze_25x25.txt")
graph=Find_the_edges(maze1)
previous_nodes, path = dijkstra_algorithm(graph, START1, GOAL1)
for i in previous_nodes:
    maze1[i[0],i[1]]=-3
for i in path:
    maze1[i[0],i[1]]=-1

plt.imshow(maze1, cmap='Pastel2')
```

```
<matplotlib.image.AxesImage at 0x13917fc40>
```



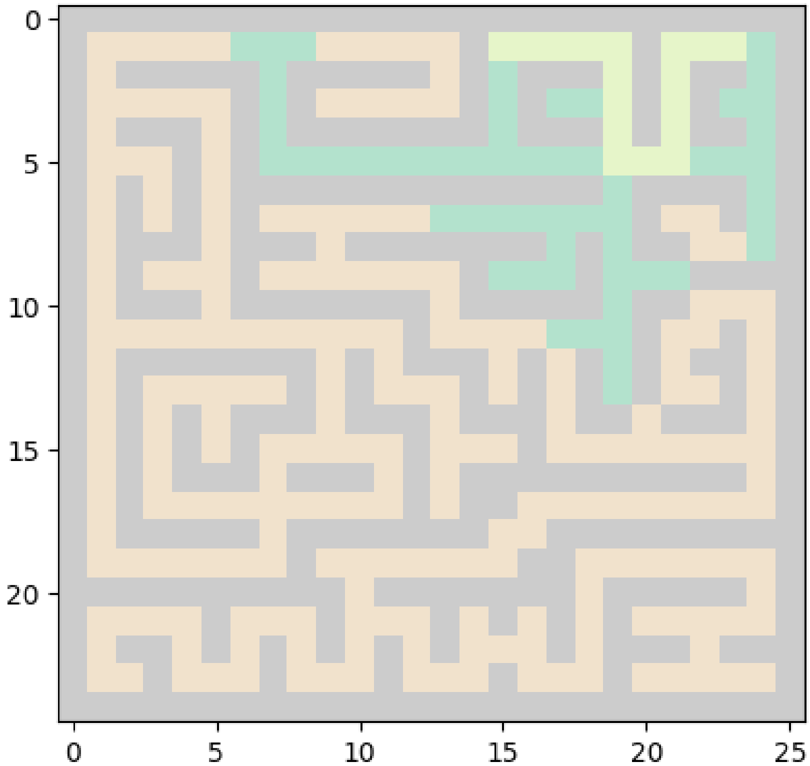

```
maze1=build_maze("my_maze_25x25.txt")
graph=Find_the_edges(maze1)
previous_nodes, path = dijkstra_algorithm(graph, START2, GOAL2)
for i in previous_nodes:
    maze1[i[0],i[1]]=-3
for i in path:
    maze1[i[0],i[1]]=-1

plt.imshow(maze1, cmap='Pastel2')
```

```
<matplotlib.image.AxesImage at 0x1391cfb80>
```




```
maze1=build_maze("my_maze_25x25.txt")
graph=Find_the_edges(maze1)
previous_nodes, path = dijkstra_algorithm(graph, START3, GOAL3)
for i in previous_nodes:
    maze1[i[0],i[1]]=-3
for i in path:
    maze1[i[0],i[1]]=-1

plt.imshow(maze1, cmap='Pastel2')
```

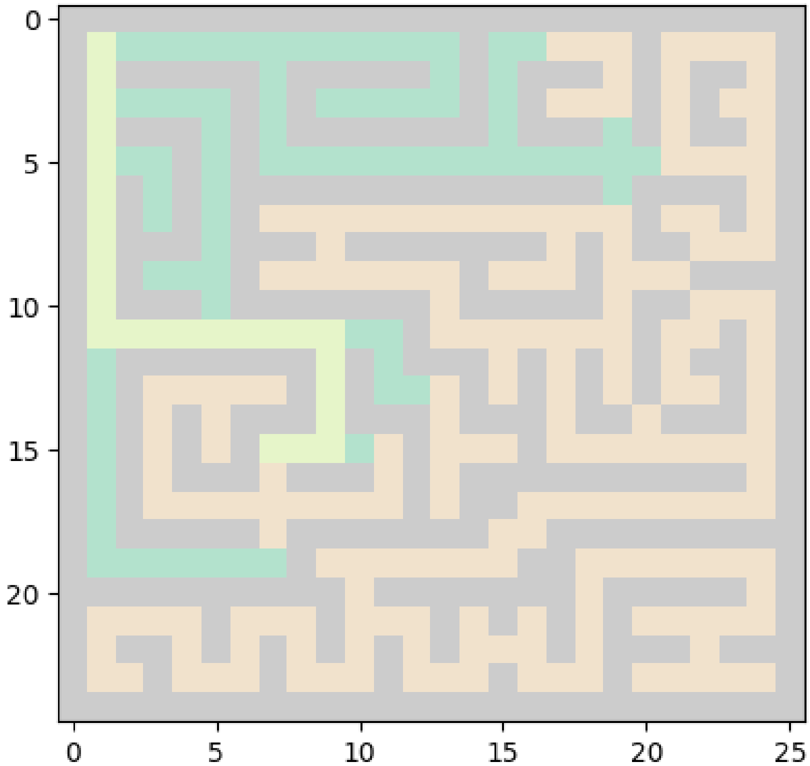
```
<matplotlib.image.AxesImage at 0x139234520>
```



BFS

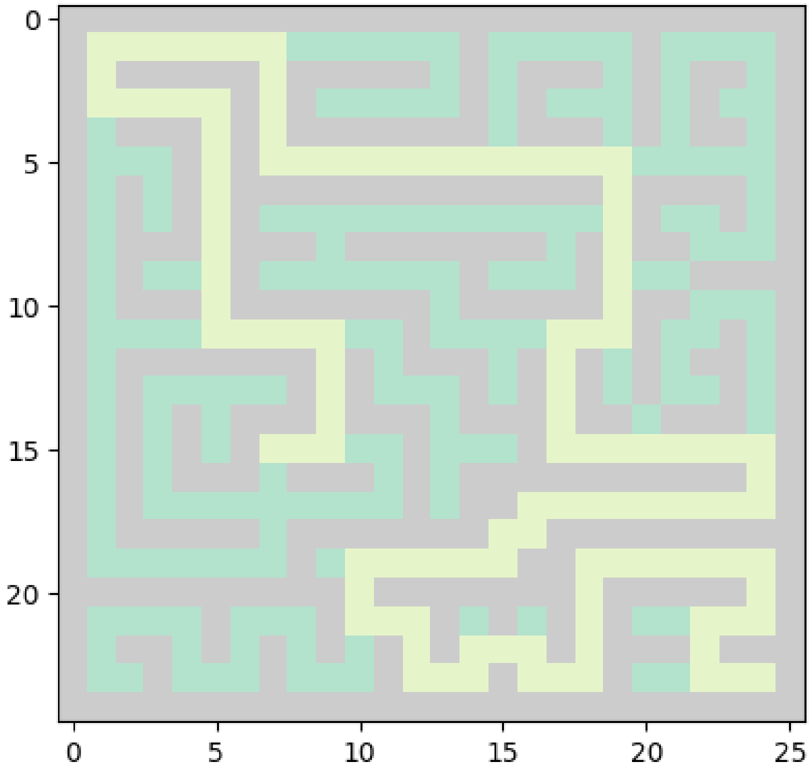
```
maze1=build_maze("my_maze_25x25.txt")
graph=Find_the_edges(maze1)
visited, path = BreadthFirst(graph, START1, GOAL1)
for i in visited:
    maze1[i[0],i[1]]=-3
for i in path:
    maze1[i[0],i[1]]=-1

plt.imshow(maze1, cmap='Pastel2')
```

```
<matplotlib.image.AxesImage at 0x139294280>
```



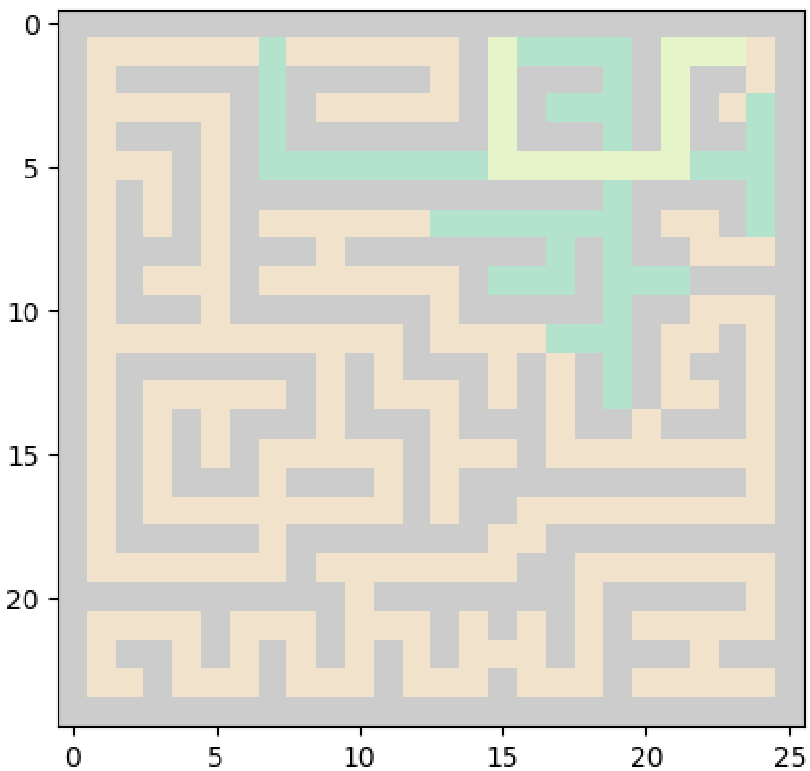
```
maze1=build_maze("my_maze_25x25.txt")
graph=Find_the_edges(maze1)
visited, path = BreadthFirst(graph, START2, GOAL2)
for i in visited:
    maze1[i[0],i[1]]=-3
for i in path:
    maze1[i[0],i[1]]=-1

plt.imshow(maze1, cmap='Pastel2')
```

```
<matplotlib.image.AxesImage at 0x1392e6be0>
```




```
maze1=build_maze("my_maze_25x25.txt")
graph=Find_the_edges(maze1)
visited, path = BreadthFirst(graph, START3, GOAL3)
for i in visited:
    maze1[i[0],i[1]]=-3
for i in path:
    maze1[i[0],i[1]]=-1

plt.imshow(maze1, cmap='Pastel2')
```

```
<matplotlib.image.AxesImage at 0x1393565b0>
```



Task - 8

The initally assumation which we made in the Find_the_edges() is the robot can only move in UP, DOWN, LEFT and RIGHT. Now it can move diagonally as well. Modifiy the function and repeat the tasks 1-6 (and visualize the solution for each). Use W=1 in this task (non need in **"Chose a broad set of values for W to see the difference"**). Explain your observations

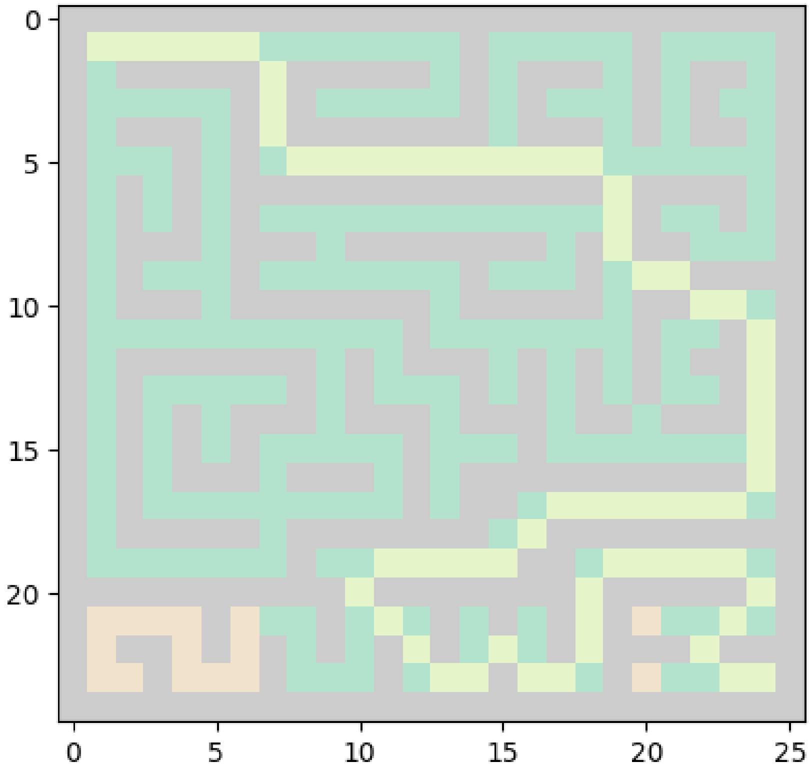
```
def Find_the_edges_2(maze):
    '''
    para1: numpy array of the maze structure
    return graph of the connected nodes
    '''

    graph={}
    directions = [(-1,0), (1,0), (0,-1), (0,1), (-1,-1), (-1,1), (1,-
1),(1,1)]

    valid = lambda x,y : 0<=x<len(maze[0]) and 0<=y<len(maze)

    for i in range(len(maze)):
        for j in range(len(maze[0])):
            if(maze[i][j]!=1):
                adj=[]
```

```
                eles=[]

                for a,b in directions:
                    if valid(i+a, j+b):
                        eles.append((i+a, j+b))

                for ele in eles:
                    if maze[ele[0]][ele[1]] == 0 or maze[ele[0]]
[ele[1]]=='3' :
                        adj.append((ele[0],ele[1]))
                graph[(i,j)] = adj
    return graph

weights_euclidean, weights_manhattan = [], []
time_taken_euclidean, time_taken_manhattan = [], []
visited_nodes_euclidean, visited_nodes_manhattan = [], []
```

A* Search | W => 1 | Heuristic => Euclidean_distance

```
# A* search with Eucliedean distance as heuristic function
# visited nodes are marked by '-3', the final path is marked by '-1'.
maze1=build_maze("my_maze_25x25.txt")
graph=Find_the_edges_2(maze1)
W = 1
start_time = time.time()
came_from, path = astar_path(graph, START, GOAL, W,
Euclidean_distance)
weights_euclidean.append(W)
time_taken_euclidean.append(time.time() - start_time)
visited_nodes_euclidean.append(len(came_from))
for i in came_from:
    maze1[i[0],i[1]]=-3
for i in path:
    maze1[i[0],i[1]]=-1

plt.imshow(maze1, cmap='Pastel2')
```

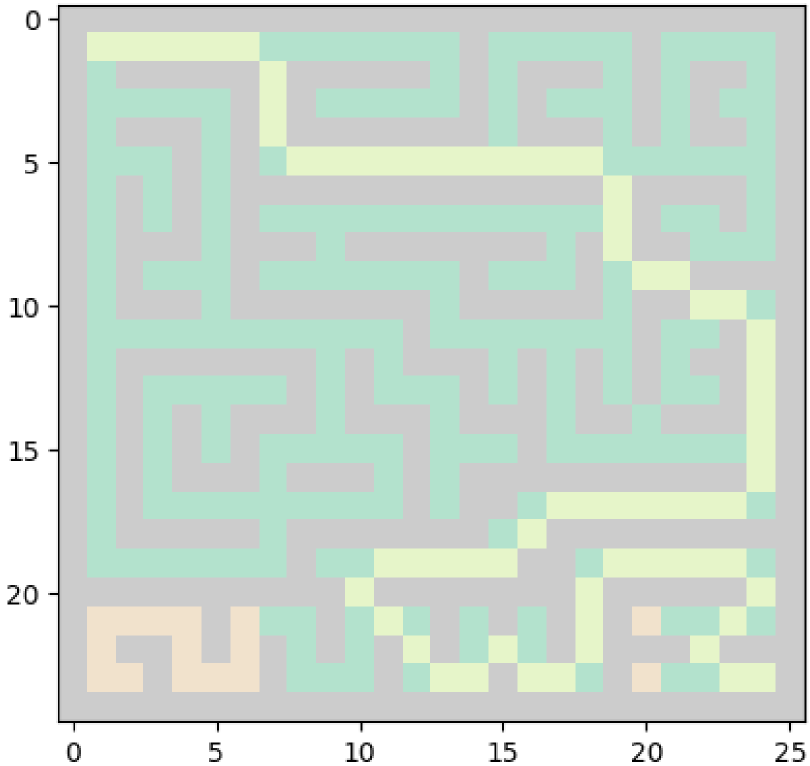
```
<matplotlib.image.AxesImage at 0x13dc3de20>
```



A* Search | W => 1 | Heuristic => Manhattan_distance

```
# A* search with Eucliedean distance as heuristic function
# visited nodes are marked by '-3', the final path is marked by '-1'.
maze1=build_maze("my_maze_25x25.txt")
graph=Find_the_edges_2(maze1)
W = 1
start_time = time.time()
came_from, path = astar_path(graph, START, GOAL, W,
Manhattan_distance)
weights_manhattan.append(W)
time_taken_manhattan.append(time.time() - start_time)
visited_nodes_manhattan.append(len(came_from))
for i in came_from:
    maze1[i[0],i[1]]=-3
for i in path:
    maze1[i[0],i[1]]=-1

plt.imshow(maze1, cmap='Pastel2')
```

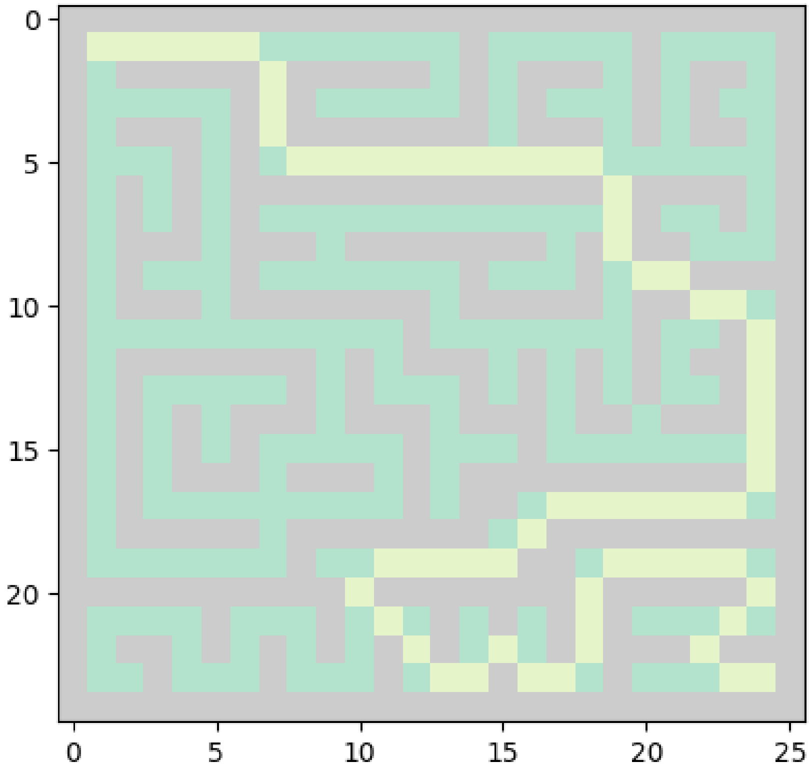
```
<matplotlib.image.AxesImage at 0x13dc3fe80>
```



A* Search | W => 0.1 | Heuristic => Euclidean_distance

```
# A* search with Eucliedean distance as heuristic function
# visited nodes are marked by '-3', the final path is marked by '-1'.
maze1=build_maze("my_maze_25x25.txt")
graph=Find_the_edges_2(maze1)
W = 0.1
start_time = time.time()
came_from, path = astar_path(graph, START, GOAL, W,
Euclidean_distance)
weights_euclidean.append(W)
time_taken_euclidean.append(time.time() - start_time)
visited_nodes_euclidean.append(len(came_from))
for i in came_from:
    maze1[i[0],i[1]]=-3
for i in path:
    maze1[i[0],i[1]]=-1

plt.imshow(maze1, cmap='Pastel2')
```

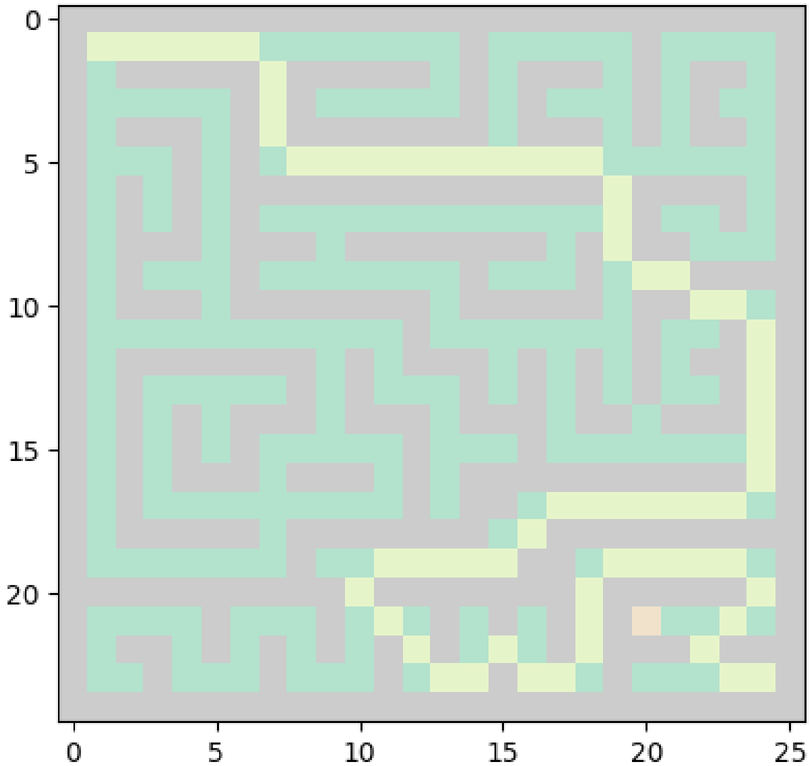
```
<matplotlib.image.AxesImage at 0x13dc78d90>
```



A* Search | W => 0.3 | Heuristic => Euclidean_distance

```
# A* search with Euclidean distance as heuristic function
# visited nodes are marked by '-3', the final path is marked by '-1'.
maze1=build_maze("my_maze_25x25.txt")
graph=Find_the_edges_2(maze1)
W = 0.3
start_time = time.time()
came_from, path = astar_path(graph, START, GOAL, W,
Euclidean_distance)
weights_euclidean.append(W)
time_taken_euclidean.append(time.time() - start_time)
visited_nodes_euclidean.append(len(came_from))
for i in came_from:
    maze1[i[0],i[1]]=-3
for i in path:
    maze1[i[0],i[1]]=-1

plt.imshow(maze1, cmap='Pastel2')
```

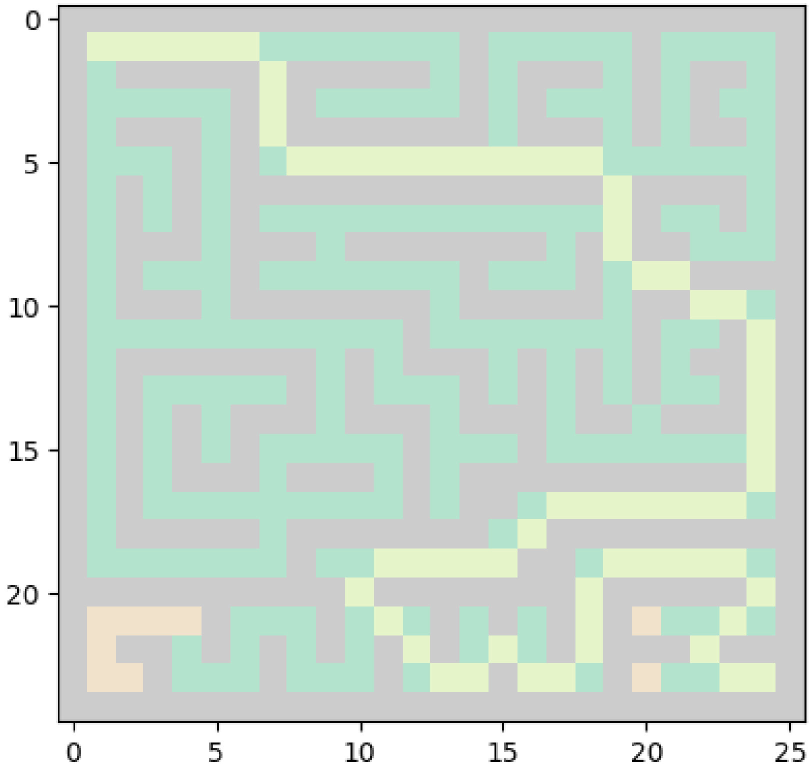
```
<matplotlib.image.AxesImage at 0x13de2e4c0>
```



A* Search | W => 0.6 | Heuristic => Euclidean_distance

```
# A* search with Eucliedean distance as heuristic function
# visited nodes are marked by '-3', the final path is marked by '-1'.
maze1=build_maze("my_maze_25x25.txt")
graph=Find_the_edges_2(maze1)
W = 0.6
start_time = time.time()
came_from, path = astar_path(graph, START, GOAL, W,
Euclidean_distance)
weights_euclidean.append(W)
time_taken_euclidean.append(time.time() - start_time)
visited_nodes_euclidean.append(len(came_from))
for i in came_from:
    maze1[i[0],i[1]]=-3
for i in path:
    maze1[i[0],i[1]]=-1

plt.imshow(maze1, cmap='Pastel2')
```

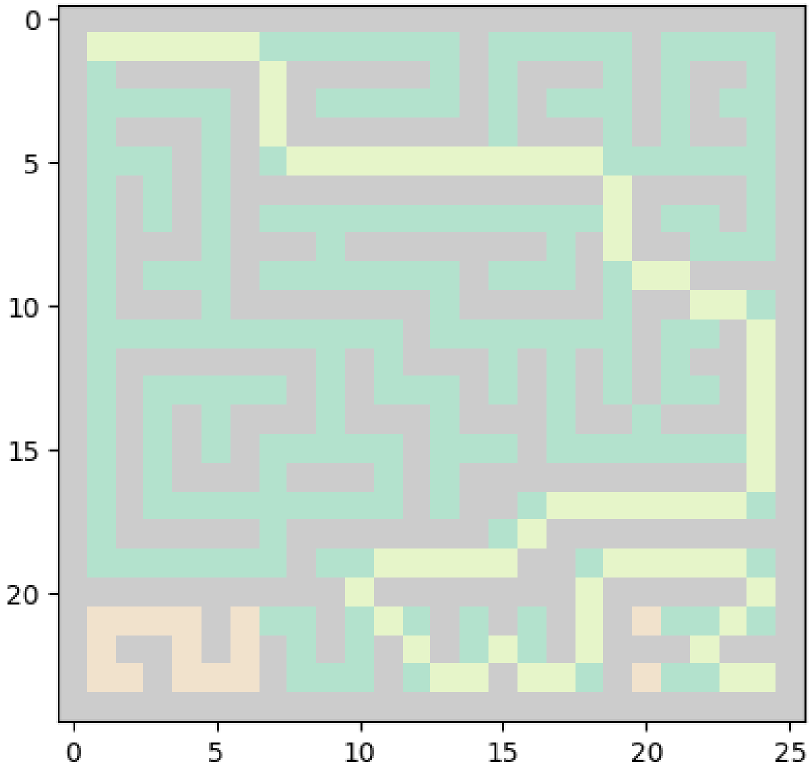
```
<matplotlib.image.AxesImage at 0x13de89670>
```



A* Search | W => 0.9 | Heuristic => Euclidean_distance

```
# A* search with Eucliedean distance as heuristic function
# visited nodes are marked by '-3', the final path is marked by '-1'.
maze1=build_maze("my_maze_25x25.txt")
graph=Find_the_edges_2(maze1)
W = 0.9
start_time = time.time()
came_from, path = astar_path(graph, START, GOAL, W,
Euclidean_distance)
weights_euclidean.append(W)
time_taken_euclidean.append(time.time() - start_time)
visited_nodes_euclidean.append(len(came_from))
for i in came_from:
    maze1[i[0],i[1]]=-3
for i in path:
    maze1[i[0],i[1]]=-1

plt.imshow(maze1, cmap='Pastel2')
```

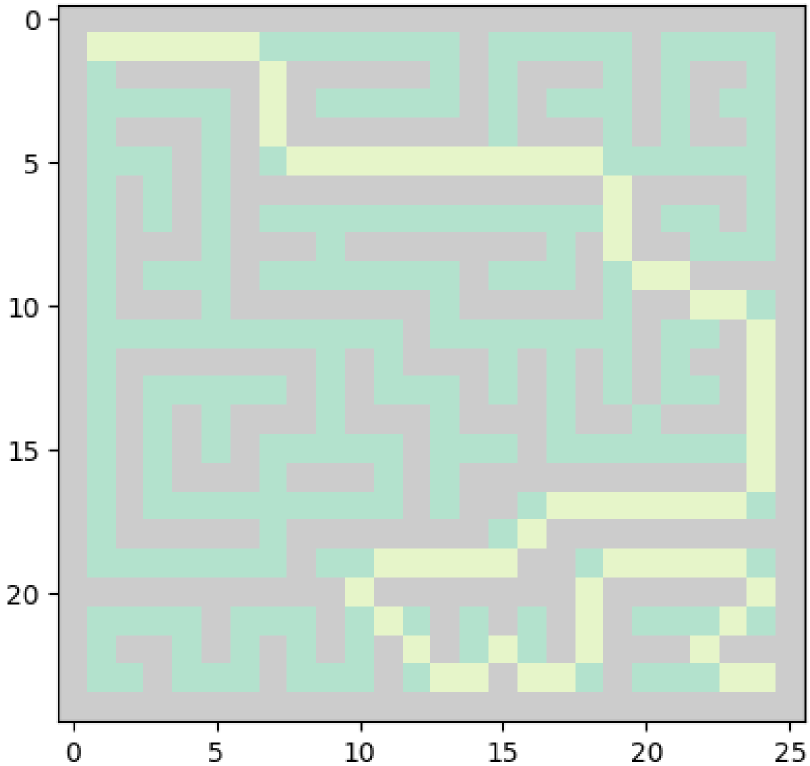
```
<matplotlib.image.AxesImage at 0x13deeb400>
```



A* Search | W => 0.1 | Heuristic => Manhattan_distance

```
# A* search with Eucliedean distance as heuristic function
# visited nodes are marked by '-3', the final path is marked by '-1'.
maze1=build_maze("my_maze_25x25.txt")
graph=Find_the_edges_2(maze1)
W = 0.1
start_time = time.time()
came_from, path = astar_path(graph, START, GOAL, W,
Manhattan_distance)
weights_manhattan.append(W)
time_taken_manhattan.append(time.time() - start_time)
visited_nodes_manhattan.append(len(came_from))
for i in came_from:
    maze1[i[0],i[1]]=-3
for i in path:
    maze1[i[0],i[1]]=-1

plt.imshow(maze1, cmap='Pastel2')
```

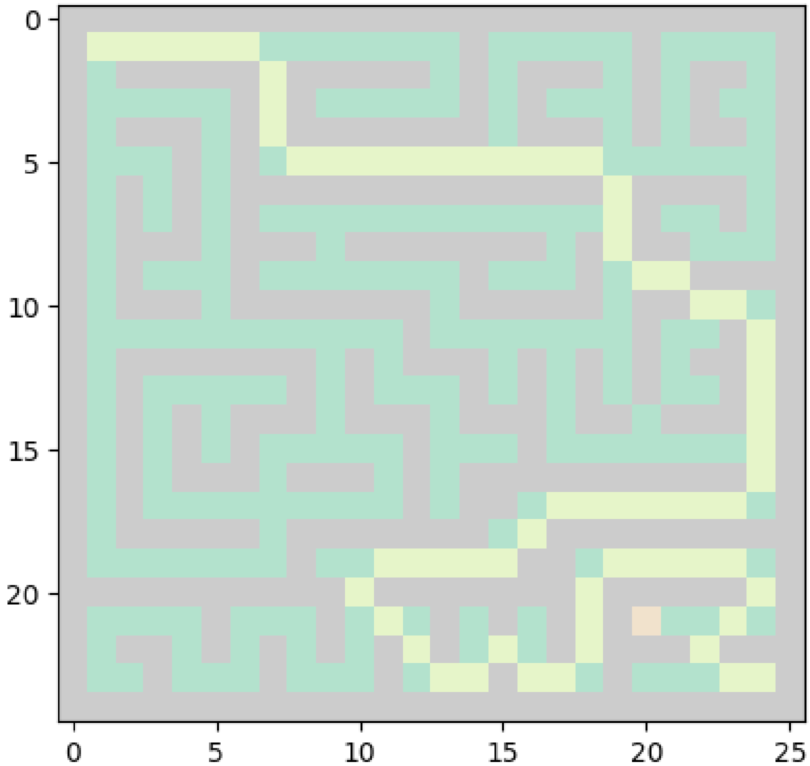
```
<matplotlib.image.AxesImage at 0x13df47670>
```



A* Search | W => 0.3 | Heuristic => Manhattan_distance

```
# A* search with Eucliedean distance as heuristic function
# visited nodes are marked by '-3', the final path is marked by '-1'.
maze1=build_maze("my_maze_25x25.txt")
graph=Find_the_edges_2(maze1)
W = 0.3
start_time = time.time()
came_from, path = astar_path(graph, START, GOAL, W,
Manhattan_distance)
weights_manhattan.append(W)
time_taken_manhattan.append(time.time() - start_time)
visited_nodes_manhattan.append(len(came_from))
for i in came_from:
    maze1[i[0],i[1]]=-3
for i in path:
    maze1[i[0],i[1]]=-1

plt.imshow(maze1, cmap='Pastel2')
```

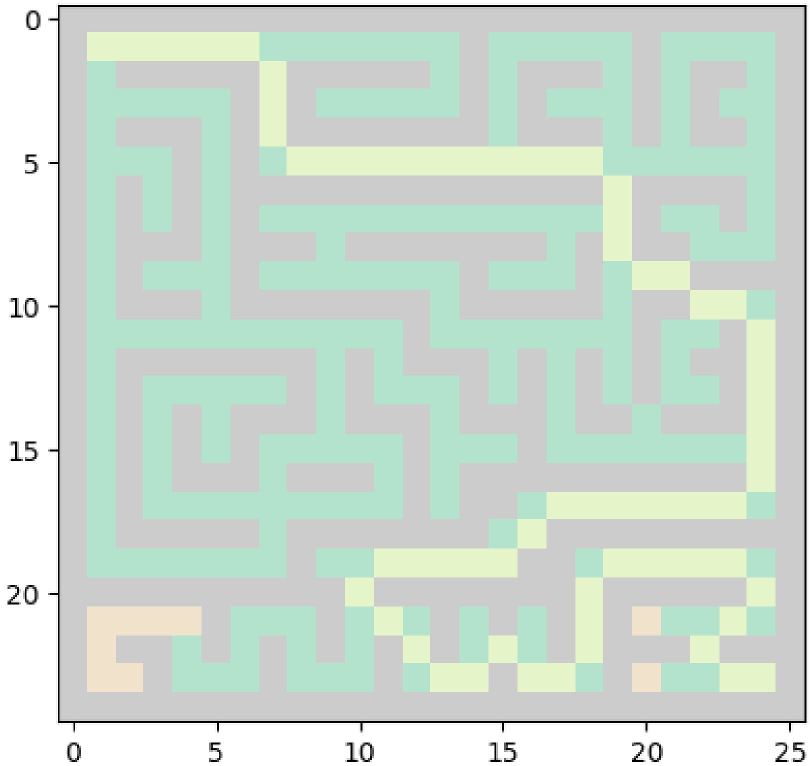
```
<matplotlib.image.AxesImage at 0x13dfa6670>
```



A* Search | W => 0.6 | Heuristic => Manhattan_distance

```
# A* search with Eucliedean distance as heuristic function
# visited nodes are marked by '-3', the final path is marked by '-1'.
maze1=build_maze("my_maze_25x25.txt")
graph=Find_the_edges_2(maze1)
W = 0.6
start_time = time.time()
came_from, path = astar_path(graph, START, GOAL, W,
Manhattan_distance)
weights_manhattan.append(W)
time_taken_manhattan.append(time.time() - start_time)
visited_nodes_manhattan.append(len(came_from))
for i in came_from:
    maze1[i[0],i[1]]=-3
for i in path:
    maze1[i[0],i[1]]=-1

plt.imshow(maze1, cmap='Pastel2')
```

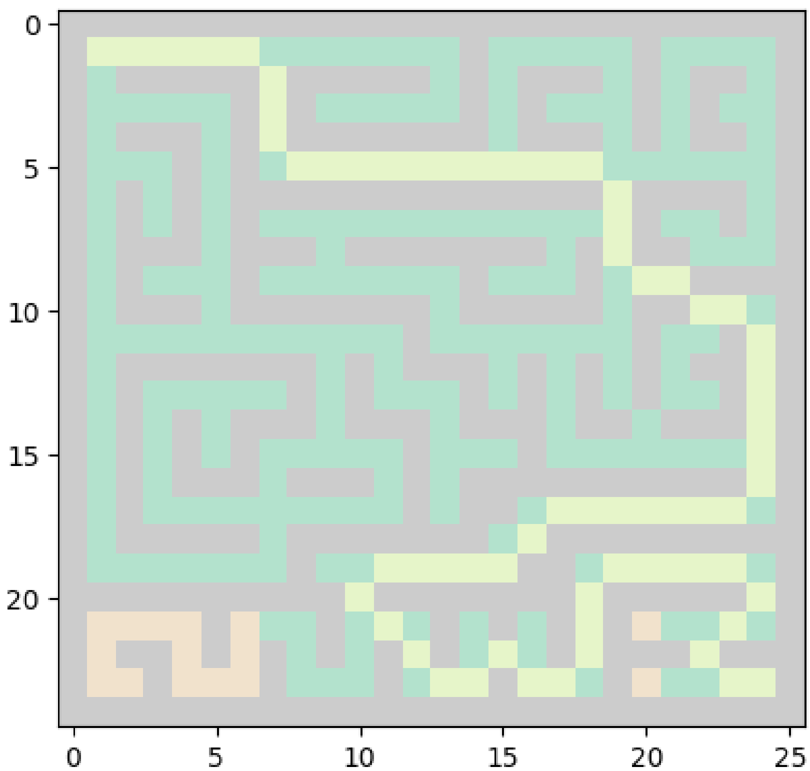
```
<matplotlib.image.AxesImage at 0x13e005640>
```



A* Search | W => 0.9 | Heuristic => Manhattan_distance

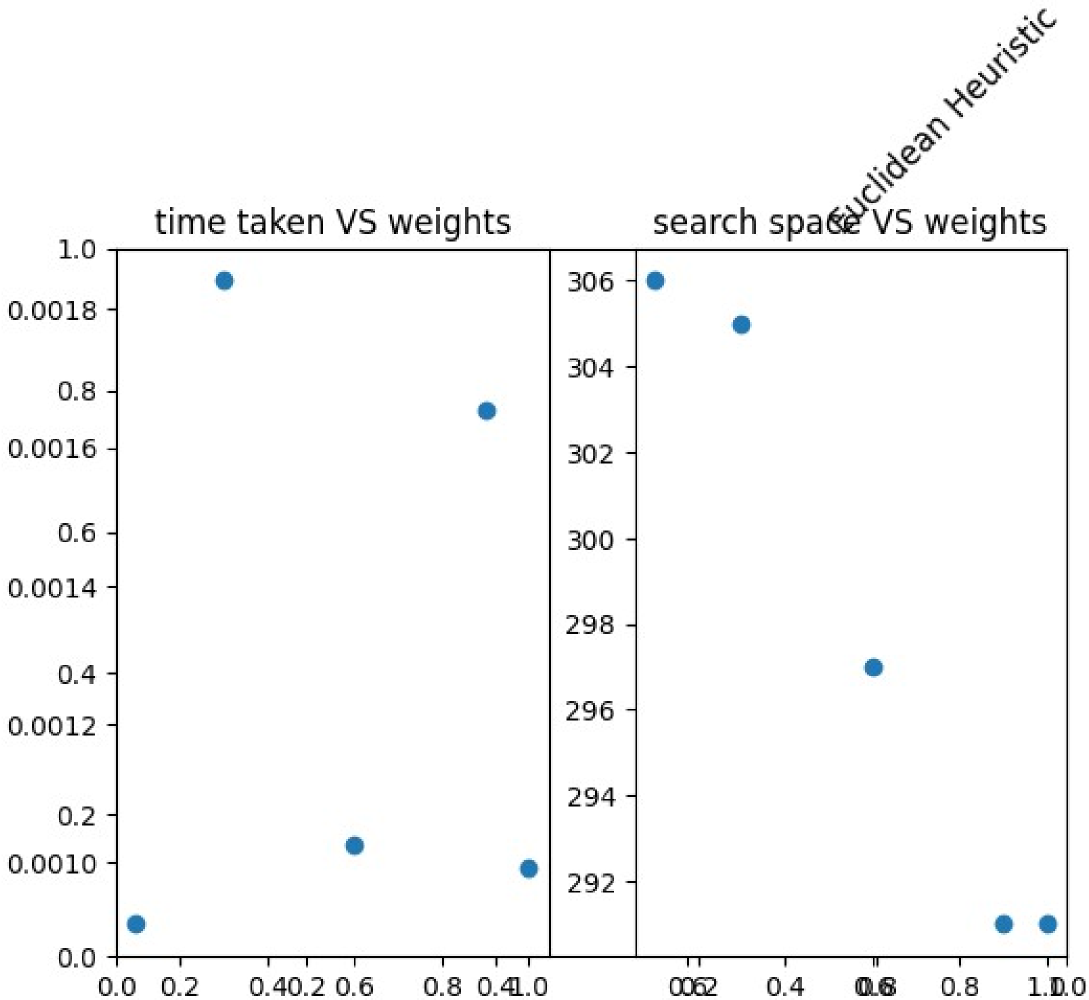
```
# A* search with Eucliedean distance as heuristic function
# visited nodes are marked by '-3', the final path is marked by '-1'.
maze1=build_maze("my_maze_25x25.txt")
graph=Find_the_edges_2(maze1)
W = 0.9
start_time = time.time()
came_from, path = astar_path(graph, START, GOAL, W,
Manhattan_distance)
weights_manhattan.append(W)
time_taken_manhattan.append(time.time() - start_time)
visited_nodes_manhattan.append(len(came_from))
for i in came_from:
    maze1[i[0],i[1]]=-3
for i in path:
    maze1[i[0],i[1]]=-1

plt.imshow(maze1, cmap='Pastel2')
```

```
<matplotlib.image.AxesImage at 0x13dc78e20>
```



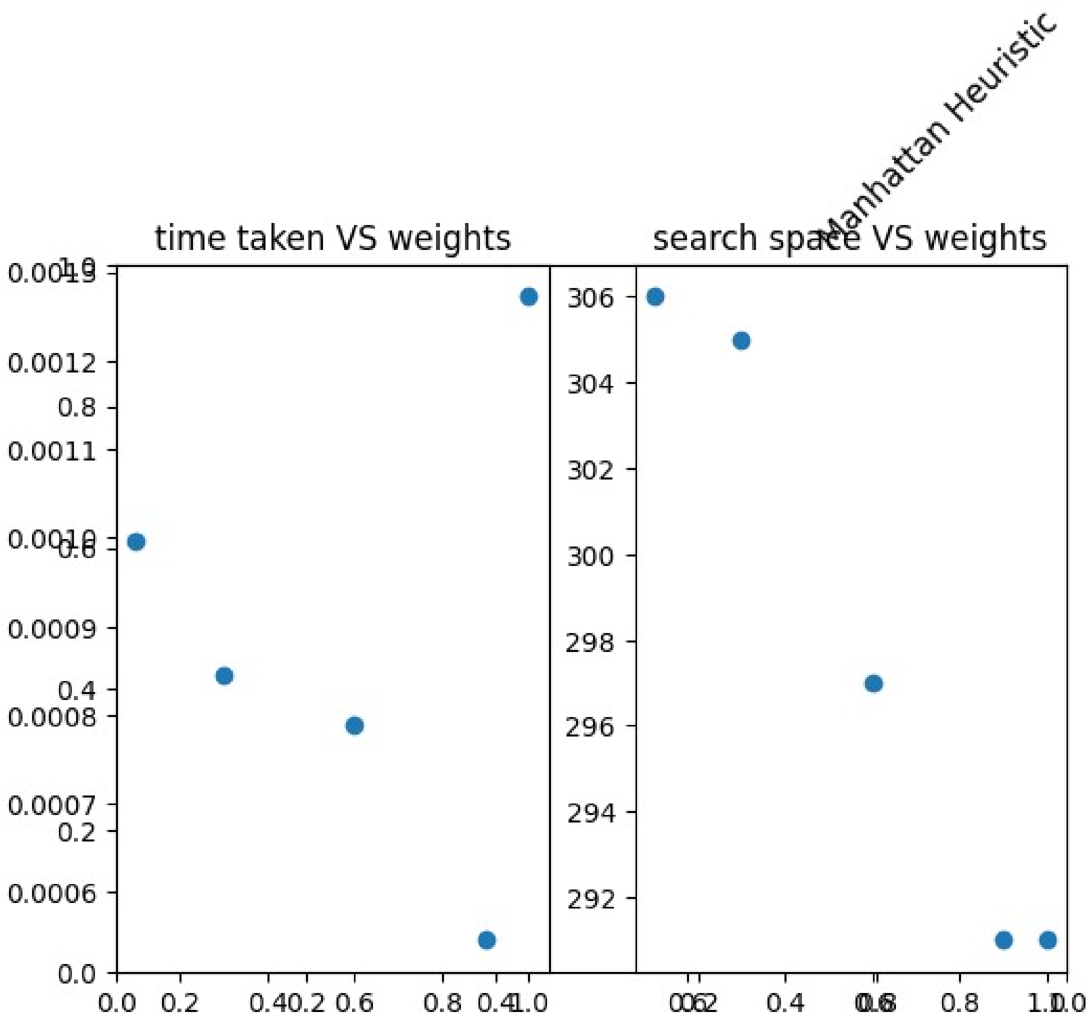
For Euclidean Heuristic function

```
fig = plt.figure()
plt.title(label="Euclidean Heuristic", loc="right", rotation = 45)
fig.add_subplot(121, title = "time taken VS weights")
plt.scatter(weights_euclidean, time_taken_euclidean)
fig.add_subplot(122, title = "search space VS weights")
plt.scatter(weights_euclidean, visited_nodes_euclidean)
plt.show()
```



For Manhattan Heuristic function

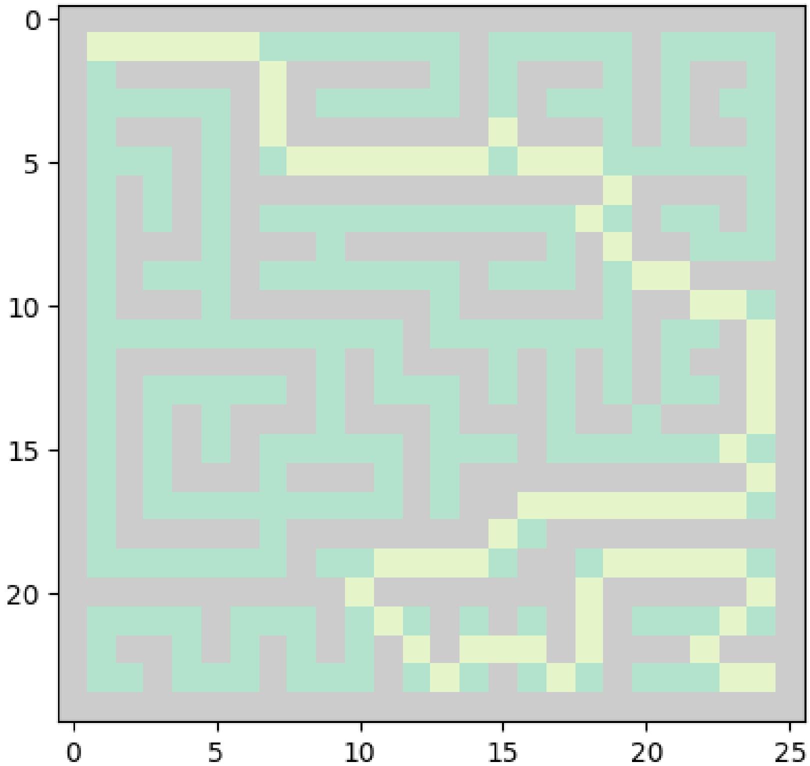
```
fig = plt.figure()
plt.title(label="Manhattan Heuristic", loc="right", rotation = 45)
fig.add_subplot(121, title = "time taken VS weights")
plt.scatter(weights_manhattan, time_taken_manhattan)
fig.add_subplot(122, title = "search space VS weights")
plt.scatter(weights_manhattan, visited_nodes_manhattan)

plt.show()
```



Dijkstra's Algorithm

```
maze1=build_maze("my_maze_25x25.txt")
graph=Find_the_edges_2(maze1)
previous_nodes, path = dijkstra_algorithm(graph, START, GOAL)
for i in previous_nodes:
    maze1[i[0],i[1]]=-3
for i in path:
    maze1[i[0],i[1]]=-1

plt.imshow(maze1, cmap='Pastel2')
```

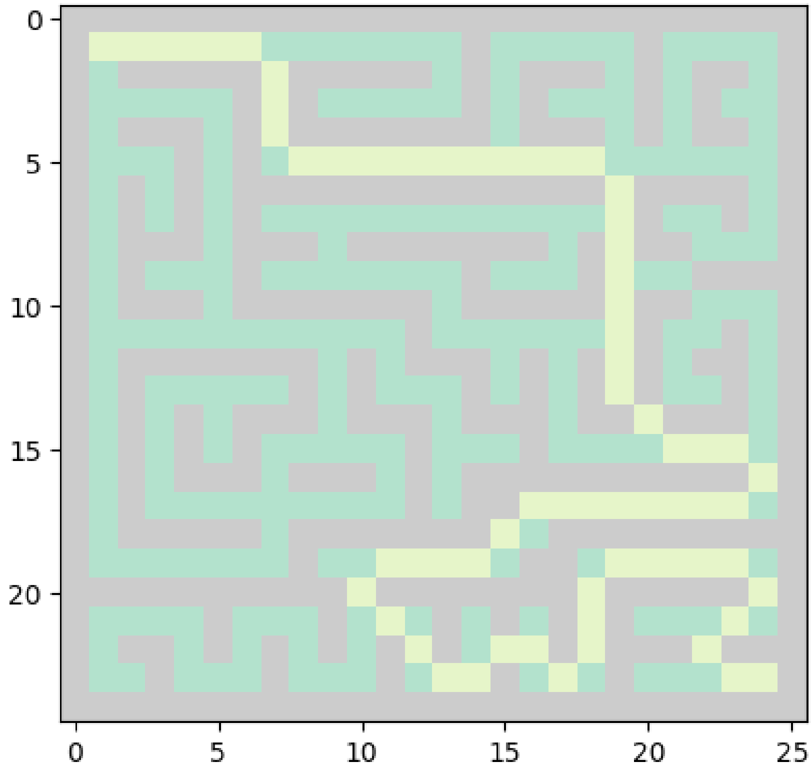
```
<matplotlib.image.AxesImage at 0x139778700>
```



BFS

```
maze1=build_maze("my_maze_25x25.txt")
graph=Find_the_edges_2(maze1)
previous_nodes, path = BreadthFirst(graph, START, GOAL)
for i in previous_nodes:
    maze1[i[0],i[1]]=-3
for i in path:
    maze1[i[0],i[1]]=-1

plt.imshow(maze1, cmap='Pastel2')
```

```
<matplotlib.image.AxesImage at 0x13975af10>
```

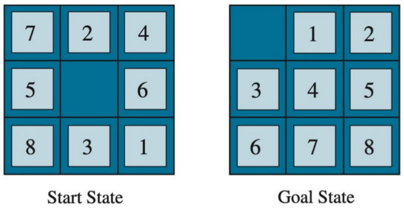


For a maze where travelling diagonal nodes is allowed for the agent, a shorter path length is required to traverse from start to goal node wrt to corresponding path using the same algorithm when travelling to diagonal nodes is not allowed.

Bonus Task (10 pt): Solving "Sliding Tile Puzzle" with A*-Search

the initial and the final configurations are given at the image below. you can use **the number of displaced tiles** as a heuristics function, h_1. Use W=1 add you code and print the optimal action sequence (which tile to move) from the intial to the final configuration.

Heuristic functions - Sliding Tile Puzzle



A typical instance of the 8-puzzle. The shortest solution is 26 actions long.

- h_1 - the number of displaced tiles (blank not included)

  h_1(Start State) $= 8$, as all the tiles are out of position
  is it admissible? YES! any tile out of place requires **at least** one move to get to the right position.

```
START_STATE = [[7,2,4],[5,0,6],[8,3,1]]
GOAL_STATE = [[0,1,2],[3,4,5],[6,7,8]]

def flatten(mat):
    return [mat[i][j] for i in range(len(mat)) for j in
range(len(mat[0]))]

def displaced_tile_heuristic(mat):
    # return number of tiles not in goal location
    return sum([1 for i,v in enumerate(flatten(mat)) if i!=v ])

def valid_next_states(mat):
    # return valid next states

    valid_moves = list()
    arr = flatten(mat)
    zero_idx = arr.index(0)

    vert_swap = lambda x: 0<=x<8
    hor_swap = lambda x: x//3 == zero_idx//3

    reshape = lambda x : list([x.pop(0) for j in range(3)] for i in
range(3))

    if vert_swap(zero_idx+3) :
        #swap_down
```

```python
        temp = arr.copy()
        temp[zero_idx], temp[zero_idx+3] = temp[zero_idx+3],
temp[zero_idx]
        valid_moves.append(reshape(temp))

    if vert_swap(zero_idx-3) :
        #swap_up
        temp = arr.copy()
        temp[zero_idx], temp[zero_idx-3] = temp[zero_idx-3],
temp[zero_idx]
        valid_moves.append(reshape(temp))

    if hor_swap(zero_idx+1) :
        #swap_right
        temp = arr.copy()
        temp[zero_idx], temp[zero_idx+1] = temp[zero_idx+1],
temp[zero_idx]
        valid_moves.append(reshape(temp))

    if hor_swap(zero_idx-1) :
        #swap_left
        temp = arr.copy()
        temp[zero_idx], temp[zero_idx-1] = temp[zero_idx-1],
temp[zero_idx]
        valid_moves.append(reshape(temp))

    return valid_moves

import heapq


class PuzzleNode:
    def __init__(self, state, parent=None, g_score=0, h_score=0):
        self.state = state
        self.parent = parent
        self.g_score = g_score
        self.h_score = h_score

    def f_score(self):
        return self.g_score + self.h_score

def astar_image_sliding(start, goal, W = 1):
    '''
    para1: Starting node
    para2: ending Node
    para3: W
    return1: list of visited nodes
    return2: nodes of shortest path
    '''
```

```python
    initial_node = PuzzleNode(start, g_score=0,
h_score=displaced_tile_heuristic(start))

    pq = []
    heapq.heappush(pq, initial_node)
    parent = {}
    visited = set()
    goal_node = None

    while pq:
        current_node = heapq.heappop(pq)
        if current_node.state == goal :
            goal_node = current_node
            path[current_node] = current_node.parent
            break
        visited.add(tuple(flatten(current_node.state)))
        parent[current_node] = current_node.parent

        for next_state in valid_next_states(current_node.state):
            if tuple(flatten(next_state)) in visited : continue
            next_node = PuzzleNode(next_state, current_node,
current_node.g_score + 1, displaced_tile_heuristic(next_state))

            for node in pq:
                if next_node.state == node.state and
next_node.f_score() >= node.f_score() : break
                heapq.heappush(pq, next_node)

    path = []
    current = goal_node
    while current and current.state != start:
        path.append(current.state)
        current = came_from[current]
    path.append(start)
    path.reverse()

    return came_from, path
'''
visited nodes - mark them as -3 in maze numpy array
path- mark them as -1 in maze numpy array
and Visualize the maze
'''
```

```
'\nvisited nodes - mark them as -3 in maze numpy array \npath- mark
them as -1 in maze numpy array\nand Visualize the maze\n'
```

```python
came_from, path = astar_image_sliding(START_STATE, GOAL_STATE)

print(came_from)
print(path)
```

{(1, 1): None, (2, 1): (1, 1), (1, 2): (1, 1), (1, 3): (1, 2), (3, 1): (2, 1), (3, 2): (2, 1), (3, 3): (3, 2), (4, 1): (3, 2), (3, 4): (3, 3), (3, 5): (3, 4), (4, 5): (3, 4), (5, 5): (4, 5), (6, 5): (5, 5), (7, 5): (6, 5), (8, 5): (7, 5), (9, 5): (8, 5), (9, 4): (8, 5), (10, 5): (9, 5), (11, 5): (10, 5), (11, 4): (10, 5), (11, 6): (10, 5), (11, 7): (11, 6), (11, 8): (11, 7), (11, 9): (11, 8), (12, 9): (11, 8), (13, 9): (12, 9), (11, 10): (12, 9), (14, 9): (13, 9), (15, 9): (14, 9), (15, 8): (14, 9), (15, 10): (14, 9), (15, 11): (15, 10), (16, 11): (15, 10), (17, 11): (16, 11), (17, 10): (16, 11), (17, 9): (17, 10), (11, 11): (11, 10), (12, 11): (11, 10), (13, 11): (12, 11), (13, 12): (12, 11), (13, 13): (13, 12), (14, 13): (13, 12), (15, 13): (14, 13), (15, 14): (14, 13), (15, 15): (15, 14), (16, 13): (15, 14), (17, 13): (16, 13), (15, 7): (15, 8), (16, 7): (15, 8), (17, 8): (16, 7), (9, 3): (9, 4), (17, 7): (16, 7), (17, 6): (16, 7), (18, 7): (17, 8), (1, 4): (1, 3), (11, 3): (11, 2), (19, 7): (18, 7), (19, 6): (18, 7), (1, 5): (1, 4), (5, 1): (4, 1), (5, 2): (4, 1), (5, 3): (5, 2), (6, 1): (5, 2), (6, 3): (5, 2), (7, 3): (6, 3), (1, 6): (1, 5), (1, 7): (1, 6), (2, 7): (1, 6), (3, 7): (2, 7), (1, 8): (2, 7), (4, 7): (3, 7), (5, 7): (4, 7), (5, 8): (4, 7), (5, 9): (5, 8), (5, 10): (5, 9), (5, 11): (5, 10), (5, 12): (5, 11), (5, 13): (5, 12), (5, 14): (5, 13), (5, 15): (5, 14), (4, 15): (5, 14), (5, 16): (5, 15), (5, 17): (5, 16), (5, 18): (5, 17), (5, 19): (5, 18), (4, 19): (5, 18), (6, 19): (5, 18), (7, 19): (6, 19), (5, 20): (6, 19), (7, 18): (6, 19), (8, 19): (7, 19), (9, 19): (8, 19), (9, 20): (8, 19), (9, 21): (9, 20), (10, 19): (9, 20), (8, 22): (9, 21), (10, 22): (9, 21), (11, 22): (10, 22), (10, 23): (10, 22), (11, 21): (10, 22), (10, 24): (10, 23), (11, 24): (10, 23), (12, 24): (11, 24), (13, 24): (12, 24), (14, 24): (13, 24), (15, 24): (14, 24), (15, 23): (14, 24), (16, 24): (15, 24), (17, 24): (16, 24), (17, 23): (16, 24), (12, 21): (11, 22), (15, 22): (15, 21), (17, 22): (17, 23), (13, 21): (12, 21), (13, 22): (12, 21), (14, 20): (13, 19), (11, 19): (10, 19), (11, 18): (10, 19), (7, 22): (8, 22), (8, 23): (8, 22), (7, 21): (8, 22), (12, 19): (11, 19), (8, 24): (8, 23), (7, 24): (6, 24), (13, 19): (12, 19), (15, 20): (14, 20), (15, 19): (14, 20), (15, 21): (14, 20), (15, 18): (14, 17), (5, 21): (5, 20), (4, 21): (5, 20), (7, 17): (7, 18), (8, 17): (7, 18), (5, 22): (5, 21), (5, 23): (5, 22), (7, 1): (6, 1), (5, 24): (5, 23), (4, 24): (5, 23), (6, 24): (5, 23), (8, 1): (7, 1), (1, 9): (1, 8), (3, 15): (4, 15), (9, 1): (8, 1), (11, 17): (11, 18), (12, 17): (11, 18), (17, 21): (17, 22), (1, 10): (1, 9), (10, 1): (9, 1), (1, 11): (1, 10), (11, 1): (10, 1), (11, 2): (10, 1), (12, 1): (11, 2), (1, 12): (1, 11), (1, 13): (1, 12), (2, 13): (1, 12), (3, 13): (2, 13), (3, 12): (2, 13), (3, 19): (4, 19), (3, 18): (4, 19), (13, 1): (12, 1), (17, 5): (17, 6), (14, 1): (13, 1), (3, 11): (3, 12), (3, 21): (4, 21), (9, 17): (8, 17), (7, 16): (8, 17), (9, 16): (8, 17), (15, 1): (14, 1), (19, 5): (19, 4), (16, 1): (15, 1), (17, 1): (16, 1), (3, 24): (4, 24), (3, 23): (4, 24), (18, 1): (17, 1), (13, 17): (12, 17), (11, 16): (12, 17), (19, 1): (18, 1), (19, 2): (18, 1), (19, 3): (19, 2), (19, 4): (19, 3), (14, 17): (13, 17), (15, 17): (14, 17), (9, 15): (9, 16), (2, 15): (3, 15), (17, 20): (17, 21), (2, 19): (3, 19), (17, 4): (17, 5), (3, 10): (3, 11), (2, 21): (3, 21), (2, 24): (3, 24), (3,

```
17): (3, 18), (7, 15): (7, 16), (1, 15): (2, 15), (1, 16): (2, 15),
(11, 15): (11, 16), (12, 15): (11, 16), (17, 19): (17, 20), (1, 19):
(1, 18), (1, 18): (1, 17), (17, 3): (17, 4), (16, 3): (17, 4), (1,
21): (2, 21), (1, 22): (2, 21), (3, 9): (3, 10), (1, 24): (1, 23), (1,
23): (1, 22), (1, 17): (1, 16), (13, 15): (12, 15), (11, 14): (12,
15), (7, 14): (7, 15), (17, 18): (17, 19), (15, 3): (16, 3), (7, 13):
(7, 14), (11, 13): (11, 14), (10, 13): (11, 14), (17, 17): (17, 18),
(14, 3): (15, 3), (7, 12): (7, 13), (17, 16): (17, 17), (18, 16): (17,
17), (9, 13): (10, 13), (9, 12): (10, 13), (18, 15): (18, 16), (19,
15): (18, 16), (13, 3): (14, 3), (13, 4): (14, 3), (7, 11): (7, 12),
(19, 14): (19, 15), (13, 5): (13, 4), (14, 5): (13, 4), (15, 5): (14,
5), (13, 6): (14, 5), (13, 7): (13, 6), (7, 10): (7, 11), (9, 11): (9,
12), (19, 13): (19, 14), (7, 9): (7, 10), (8, 9): (7, 10), (9, 10):
(8, 9), (19, 12): (19, 13), (9, 9): (8, 9), (7, 8): (8, 9), (9, 8):
(8, 9), (9, 7): (9, 8), (19, 11): (19, 12), (7, 7): (7, 8), (19, 10):
(19, 11), (20, 10): (19, 11), (21, 10): (20, 10), (19, 9): (20, 10),
(21, 11): (20, 10), (21, 12): (21, 11), (22, 10): (21, 11), (22, 12):
(21, 11), (23, 12): (22, 12), (23, 13): (22, 12), (23, 14): (23, 13),
(22, 14): (23, 13), (22, 15): (23, 14), (21, 14): (22, 14), (22, 16):
(22, 15), (21, 16): (22, 15), (23, 16): (22, 15), (23, 17): (23, 16),
(23, 18): (23, 17), (22, 18): (23, 17), (21, 18): (22, 18), (23, 10):
(22, 10), (23, 9): (22, 10), (20, 18): (21, 18), (23, 8): (23, 9),
(22, 8): (23, 9), (19, 18): (20, 18), (19, 19): (20, 18), (19, 20):
(19, 19), (19, 21): (19, 20), (19, 22): (19, 21), (21, 8): (22, 8),
(21, 7): (22, 8), (19, 23): (19, 22), (19, 24): (19, 23), (20, 24):
(19, 23), (21, 24): (20, 24), (21, 23): (20, 24), (21, 22): (21, 23),
(22, 22): (21, 23), (23, 22): (22, 22), (21, 21): (22, 22), (23, 21):
(22, 22), (23, 23): (22, 22), (23, 24): (23, 23)}
[[[7, 2, 4], [5, 0, 6], [8, 3, 1]]]
```