Artificial Intelligence and Machine Learning Lab 3

SHRIA SRIVASTAVA

Class: TE IT

BATCH: D

UID: 2019140064

Problem Statement:

Implementation of a Tower Defense game,

there are many enemies that are all headed to the same place.

you can place towers anywhere, and they act as obstacles that affect the paths taken by enemies

Implement the above strategy using A* algorithm both.

-Show the status of Frontier at every stage

-Show the final solution path the enemies have taken to reach to the destination

- Implement heuristic Function for A* and show the Nodes in Frontier at each stage with F() value

- Show analysis of both the algorithms with respect to

  1-total No. of nodes getting generated

  2- Optimal solution

  3- Time Complexity and Space Complexity

Program:

```
import math
#Node class
class Node:
    parent = None
```

```python
    f = 0
    g = 0
    h = 0
    def __init__(self, x, y, parent) -> None:
        self.x = x
        self.y = y
        self.parent = parent

#User input for grid
s = int(input("Enter the side length of the grid: "))
o = int(input("Enter the no of obstacles: "))
maze = [[0 for i in range(s)] for j in range(s)]
for _ in range(o):
    i, j = [int(x) for x in input("Enter the coordinates of obstacles: ").split()]
    maze[i][j] = 1

# closed and open lists for A* algorithm
closed = []
open = []

i, j = [int(x) for x in input("Enter the start node : ").split()]
start_node = Node(i, j, None)
i, j = [int(x) for x in input("Enter the end node : ").split()]
goal_node = Node(i, j, None)
goal_node.h = 0
open.append(start_node)

# Function to calculate heuristic from a given node
def heuristic(node):
```

```
    #Euclidian distance
    return math.sqrt((goal_node.x - node.x)**2 +
(goal_node.y - node.y)**2)

#Managing newly generated node
def manage_new_node(x, y, parent_node):
    if x<0 or x>=s or y<0 or y>=s or maze[x][y] == 1:
        return -1

    global open
    n = Node(x, y, parent_node)
    n.h = heuristic(n)
    n.g = parent_node.g + math.sqrt((parent_node.x -
n.x)**2 + (parent_node.y - n.y)**2)
    n.f = n.g + n.h
    append_or_not = 1
    if n.x == goal_node.x and n.y == goal_node.y:#If
generated node is same as goal node
        return n
    for c in closed:
        if n.x == c.x and n.y == c.y:#If node is already
expanded
            append_or_not = 0
    for o in open:
        if n.x == o.x and n.y == o.y:
            if o.f <= n.f:
                append_or_not = 0
            else:# Node in open and new f value is
lesser
                append_or_not = 0
                o.f, o.g, o.h, o.parent = n.f, n.g, n.h,
n.parent
```

```
                open = sorted(open, key=lambda
node:(node.f, node.h))
    if append_or_not:
        open.append(n)
    open = sorted(open, key=lambda node:(node.f,
node.h))# Sort the open list after appending
    return -1


def print_path(node):#printing path using parent pointer
tracing
    path = []
    while node.parent != None:
        path.append((node.x, node.y))
        node = node.parent
    path.append((node.x, node.y))
    path.reverse()
    print("The path is:")
    print(path)

def print_node_list(ls):# Print a list of node objects
    print([(n.x, n.y) for n in ls])

def a_star(): # Main algorithm
    while len(open)!= 0:
        print("Open nodes:")
        print_node_list(open)
        curr_node = open.pop(0)
        closed.append(curr_node)
        x = curr_node.x
        y = curr_node.y
        #Nodes
```

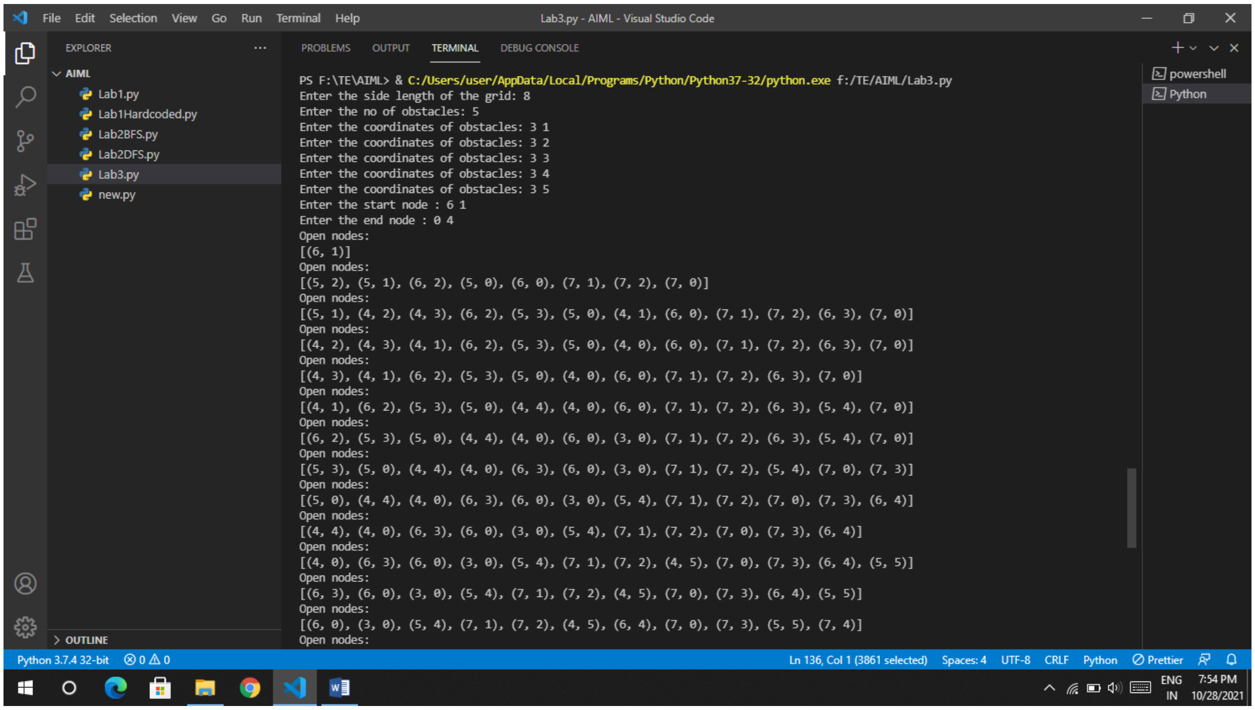
```
#L
n = manage_new_node(x, y-1, curr_node)
if n!= -1:
    print_path(n)
    break
#U
n = manage_new_node(x-1, y, curr_node)
if n!= -1:
    print_path(n)
    break
#R
n = manage_new_node(x, y+1, curr_node)
if n!= -1:
    print_path(n)
    break
#D
n = manage_new_node(x+1, y, curr_node)
if n!= -1:
    print_path(n)
    break
#UL
n = manage_new_node(x-1, y-1, curr_node)
if n!= -1:
    print_path(n)
    break
#UR
n = manage_new_node(x-1, y+1, curr_node)
if n!= -1:
    print_path(n)
    break
#LL
n = manage_new_node(x+1, y-1, curr_node)
```

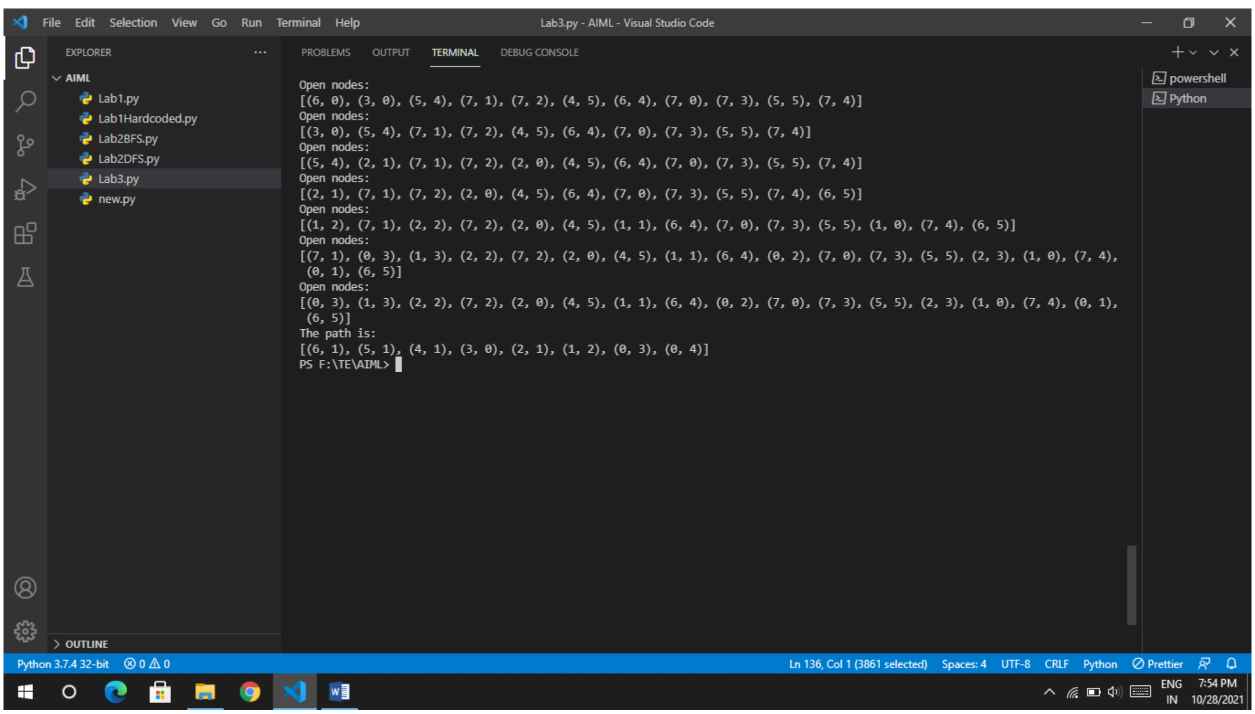
```
        if n!= -1:
            print_path(n)
            break
        #LR
        n = manage_new_node(x+1, y+1, curr_node)
        if n!= -1:
            print_path(n)
            break

# Final driver code
start_node.h = heuristic(start_node)
start_node.f = start_node.g + start_node.h


a_star()
```

Output:





Conclusion:

A* is a modification on BFS algorithm. It uses heuristics along with previously incurred cost to determine the shortest path by generating

lesser nodes than normal BFS. This algorithm is best for simple pathfinding applications like mazes.