RA1911026010029
Thejaswin S

# EXP-5 Developing Best first search and A* Algorithm for real-world problems

## AIM

To implement a program for path-finding in the maze using Best first Search and A* algorithm.

### Problem

This problem involves a highly simplified version of path-finding taking terrain into account. You are given an n × n grid (with co-ordinates in the range 0, . . . ,(n − 1)), a starting position on the grid, and a goal location. Each point on the grid is assigned an integer, giving its elevation. It is possible to move from a location to any of its 4 neighbors (except at a boundary). The cost of such a move is 1 + the absolute value of the difference in elevation.
However, if the difference between two points is ≥ 4, this represents an impassible cliff: the agent can't go "up" a cliff (i.e. move from the lower-ranked point to the higher), and going "down" will wreck the agent. The goal is to determine the least-cost path from the start location to the goal location. The following is an example; conventionally the top left-hand corner is (0, 0) and the lower right corner is (n − 1, n − 1).

### Best First Search

Best first search is a traversal technique that decides which node is to be visited next by checking which node is the most promising one and then checking it. For this, it uses an evaluation function to decide the traversal.

This best first search technique of tree traversal comes under the category of heuristic search or informed search technique.

The cost of nodes is stored in a priority queue. This makes the implementation of best-first search is same as that of breadth First search. We will use the priority queue just like we use a queue for BFS.

### Algorithm

**f(n) = h(n)**

1. Retrieve the start node and end node. Put start node into open list

2. If the open list is empty, the search fails, and the end node is not reachable. If the open list is not empty,

retrieve the node with the smallest heuristic value, denoted as current.

3. Get neighbors of current. For each neighbor, if the neighbor is on the open list or closed

list, ignore this neighbor. Otherwise set neighbor’s parent as current, put the neighbor

into the open list.

4. If the current is not the end node, return to step 2.

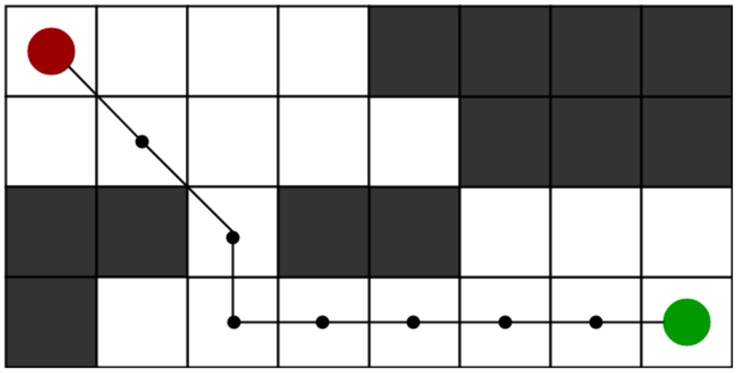
5. If current == end node, get the parent of the end node. Then get the parent of the parent

(repeat this process until no parent node is found). All the parents construct a path.

This algorithm will traverse the shortest path first in the queue. The time complexity of the algorithm is given by O(n*logn).

## A* Search Algorithm

A* Search algorithm is one of the best and most popular techniques used in path-finding and graph traversals. It is a handy algorithm that is often used for map traversal to find the shortest path to be taken. A* was initially designed as a graph traversal problem, to help build a robot that can find its own course. It still remains a widely popular algorithm for graph traversal.
It searches for shorter paths first, thus making it an optimal and complete algorithm. An optimal algorithm will find the least cost outcome for a problem, while a complete algorithm finds all the possible outcomes of a problem.
A major drawback of the algorithm is its space and time complexity. It takes a large amount of space to store all possible paths and a lot of time to find them.
**f(n) = g(n) + h(n)**
g(n) = shows the shortest path’s value from the starting node to node n
h(n) = The heuristic approximation of the value of the node
The f(n) denotes the cost, A* chooses the node with the lowest f(n) value



**Heuristic Function**
The chosen heuristics is Manhattan Distance. The formula of Manhattan Distance between node n and endpoint e is
$H(n) = Abs(n.x - e.x) + Abs(n.y - e.y)$
The reason of using Manhattan Distance is that Manhattan Distance is the minimum distance between a node to end node. Considering a map with no elevation difference: from the start node to the end node, the distance will be H(n)

**Algorithm**

1. Retrieve start node and end node, with F(n) = 0.
2. Initialize open list and closed list. Put start node in the open list.
3. If open node is empty, search fails, and end node is not reachable. Otherwise get the
node with smallest f(n), denoted as “current”.
4. Find neighbours for f(n). For each neighbour:
a) If neighbour is in closed list. Ignore.
b) If neighbour is not in open list, calculate g(n) for neighbour by g(parent) + cost(neighbour, parent). Calculate h(n) using Manhattan Distance. Calculate and store f(n) = g(n) + h(n). Set parent of neighbour to current. Put neighbour into open list.
c) If neighbour is in open list, calculate new g(n) -> g’(n) for neighbour using g(parent), calculate new f(n) => f’(n) with f’(n) = g’(n) + h(n). If f’(n) < f(n), update neighbour’s f(n) with f’(n), update neighbour’s parent with current. Otherwise, ignore neighbour.
5. Put current into closed list, remove current from open list.
a) If current == end node, then A* reaches the goal.
b) Get the parent of the end node. Then get the parent of the parent (repeat this process until no parent node found). All the parents construct a path.
c) Otherwise, go back to step 3.

RA1911026010029
Thejaswin S

**Program**

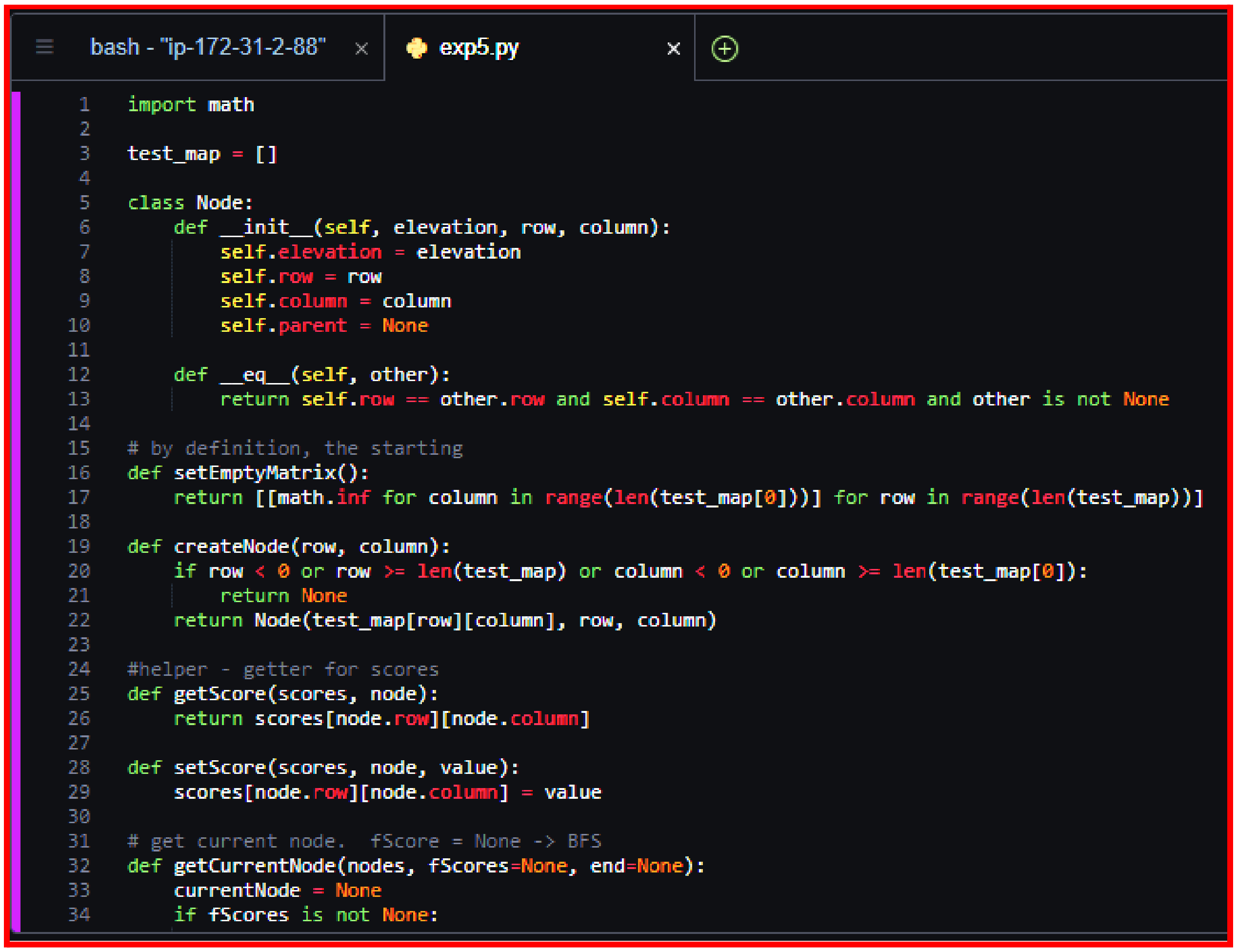
```
import math

test_map = []

class Node:
    def __init__(self, elevation, row, column):
        self.elevation = elevation
        self.row = row
        self.column = column
        self.parent = None

    def __eq__(self, other):
        return self.row == other.row and self.column == other.column and
other is not None

# by definition, the starting
def setEmptyMatrix():
    return [[math.inf for column in range(len(test_map[0]))] for row in
range(len(test_map))]

def createNode(row, column):
    if row < 0 or row >= len(test_map) or column < 0 or column >=
len(test_map[0]):
        return None
    return Node(test_map[row][column], row, column)

#helper - getter for scores
def getScore(scores, node):
    return scores[node.row][node.column]

def setScore(scores, node, value):
    scores[node.row][node.column] = value

# get current node.  fScore = None -> BFS
def getCurrentNode(nodes, fScores=None, end=None):
    currentNode = None
    if fScores is not None:
```

```python
        smallestFScore = None
        for node in nodes:
            currentFScore = getScore(fScores, node)
            if currentNode is None or currentFScore < smallestFScore:
                smallestFScore = currentFScore
                currentNode = node
    else:
        for node in nodes:
            if currentNode is None or getHeuristic(currentNode, end) > getHeuristic(node, end):
                currentNode = node
    return currentNode

def getPath(node):
    path = []
    temp = node
    while temp is not None:
        path.append(temp)
        temp = temp.parent
    return path[::-1]

def getParent(node):
    return node.parent

def getNeighbours(parent):
    neighbours = [createNode(parent.row - 1, parent.column),
        createNode(parent.row + 1, parent.column),
        createNode(parent.row, parent.column - 1),
        createNode(parent.row, parent.column + 1)]
    return [neighbour for neighbour in neighbours if neighbour is not None]

# heuristic function
def getHeuristic(current, end):
    return abs(current.row - end.row) + abs(current.column - end.column)

def readFile():
    with open("Asst1.data.txt") as f:
        lines = f.read().splitlines()
    size = int(lines[0])
    pointsStr = lines[1]
    mapRows = lines[2:]
    initMap(mapRows)
    if size != len(test_map) or size != len(test_map[0]):
        print("Size and map size does not match")
        return
    return getStartEndNodes(pointsStr)

# initialize search map
def initMap(rows):
    for row in rows:
        arr = []
        rowStr = removeArrChar(row)
        valuesStr = rowStr.split()
        for value in valuesStr:
            arr.append(int(value))
        test_map.append(arr)

# display final result in traditional (x, y) format, aka (column, row)
def writePath(nodes, start, end, mode, fScores=None, gScores=None):
    if nodes is None:
        return
    print("The path found from {0} to {1}
is".format(getCoordinateStr(start), getCoordinateStr(end)))
    if mode == "aStar":
        writeAStarNodeInfo(nodes, fScores, gScores, end)
    else:
        writeBFSNodeInfo(nodes, end)

# helper - remove extra characters for array
def removeArrChar(arrStr):
    return arrStr.replace("[", "").replace("]", "")

# helper - get elevation difference
def getElevDifference(node, aNode):
    return abs(node.elevation - aNode.elevation)

# helper - get coordinate as string
```

```
def getCoordinateStr(node):
    return "({0}, {1})".format(node.column, node.row)

# helper - write each node info for A*
def writeAStarNodeInfo(nodes, fScores, gScores, end):
    for node in nodes:
        print("Node(x, y): ", getCoordinateStr(node), getScoreInfo(node,
fScores, gScores, end))

# helper - for A* to get score info for each node
def getScoreInfo(node, fScores, gScores, end):
    fScoreStr = "FScore is: {0},
".format(fScores[node.row][node.column])
    gScoreStr = "GScore is: {0},
".format(gScores[node.row][node.column])
    hScoreStr = "HScore is: {0}".format(getHeuristic(node, end))
    return fScoreStr + gScoreStr + hScoreStr

# helper - write each node info for Best First Search
def writeBFSNodeInfo(nodes, end):
    for node in nodes:
        print("Node(x, y): ", getCoordinateStr(node), "HScore is: ",
getHeuristic(node, end))

# helper - get start and end node
def getStartEndNodes(pointsStr):
    points = removeArrChar(pointsStr)
    pointsArr = points.split()
    nodes = []
    for item in pointsArr:
        temp = item.split(",")
        nodes.append(createNode(int(temp[1]), int(temp[0])))

    start = nodes[0]
    end = nodes[1]
    return start, end

# best first search
def BFS(start, end):
    if start is not None and end is not None:
        closedNodes = []
        openNodes = [start]
        count = 0
        while len(openNodes) != 0:
            currentNode = getCurrentNode(openNodes, None, end)
            neighbours = getNeighbours(currentNode)

            for neighbour in neighbours:
                if neighbour not in openNodes and neighbour not in closedNodes:
                    if 1 + getElevDifference(neighbour, currentNode) < 4:
                        openNodes.append(neighbour)
                        neighbour.parent = currentNode

            openNodes.remove(currentNode)
            closedNodes.append(currentNode)
            count += 1

            if currentNode == end:
                print("Best First Search has found the path")
                print("There are {0} nodes expended".format(len(openNodes) + len(closedNodes)))
                print("The number of iterations is: ", count)
                return getPath(currentNode)

        print("No path found between", getCoordinateStr(start), "and", getCoordinateStr(end), "(BFS)")
        return None
    print("Start point or end point is out of map. Please check your input. Bye!")
    return None

# A* search
def aStar(start, end):
    if start is not None and end is not None:
        closedNodes = []
        openNodes = [start]
        # initialize fScore and gScores
        fScores = setEmptyMatrix()
        gScores = setEmptyMatrix()
        # initialize fScore and gScore of start
        setScore(fScores, start, 0)
        setScore(gScores, start, 0)
        count = 0
        while len(openNodes) != 0:
            currentNode = getCurrentNode(openNodes, fScores)
            if currentNode == end:
                print("A* Search has found the path")
                print("There are {0} nodes
expended".format(len(openNodes) + len(closedNodes)))
                print("The number of iterations is: ", count)
                return fScores, gScores, getPath(currentNode)

            neighbours = getNeighbours(currentNode)
            for neighbour in neighbours:
                # handle neighbours
                if neighbour in closedNodes or
getElevDifference(neighbour, currentNode) >= 4:
                    continue
                else:
                    if neighbour not in openNodes:
                        gScore = 1 + getElevDifference(neighbour,
currentNode) + getScore(gScores, currentNode)
                        openNodes.append(neighbour)
                        # set parent and scores for newly joined nodes
                        setScore(gScores, neighbour, gScore)
                        setScore(fScores, neighbour, gScore +
getHeuristic(neighbour, end))
                        neighbour.parent = currentNode
                    else:
                        gScore = getScore(gScores, neighbour)
                        newGScore = 1 + getElevDifference(neighbour,
currentNode) + getScore(gScores, currentNode)
                        if newGScore < gScore:
                            # update parent
                            neighbour.parent = currentNode
                            # update gScore and fScore
                            setScore(gScores, neighbour, newGScore)
                            setScore(fScores, neighbour, newGScore +
getHeuristic(neighbour, end))

            openNodes.remove(currentNode)
            closedNodes.append(currentNode)
            count += 1
        print("No path found between", getCoordinateStr(start), "and",
getCoordinateStr(end), "(A*)")
        return None, None, None
    print("Start point or end point is out of map. Please check your
input. Bye!")
    return None, None, None

def init():
    # initialize start and end node
    startNode, endNode = readFile()

    # print research for A* search
    fScores, gScores, path = aStar(startNode, endNode)
    writePath(path, startNode, endNode, "aStar", fScores, gScores)


print("-------------------------------------------------------------")

    # print result for BFS search
    writePath(BFS(startNode, endNode), startNode, endNode, "BFS")

init()
```

RA1911026010029
Thejaswin S

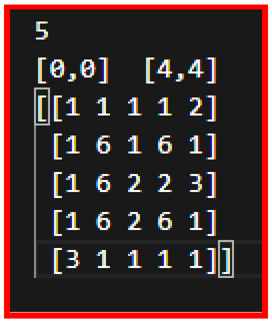
## AWS Implementation



```python
import math

test_map = []

class Node:
    def __init__(self, elevation, row, column):
        self.elevation = elevation
        self.row = row
        self.column = column
        self.parent = None

    def __eq__(self, other):
        return self.row == other.row and self.column == other.column and other is not None

# by definition, the starting
def setEmptyMatrix():
    return [[math.inf for column in range(len(test_map[0]))] for row in range(len(test_map))]

def createNode(row, column):
    if row < 0 or row >= len(test_map) or column < 0 or column >= len(test_map[0]):
        return None
    return Node(test_map[row][column], row, column)

#helper - getter for scores
def getScore(scores, node):
    return scores[node.row][node.column]

def setScore(scores, node, value):
    scores[node.row][node.column] = value

# get current node.  fScore = None -> BFS
def getCurrentNode(nodes, fScores=None, end=None):
    currentNode = None
    if fScores is not None:
```

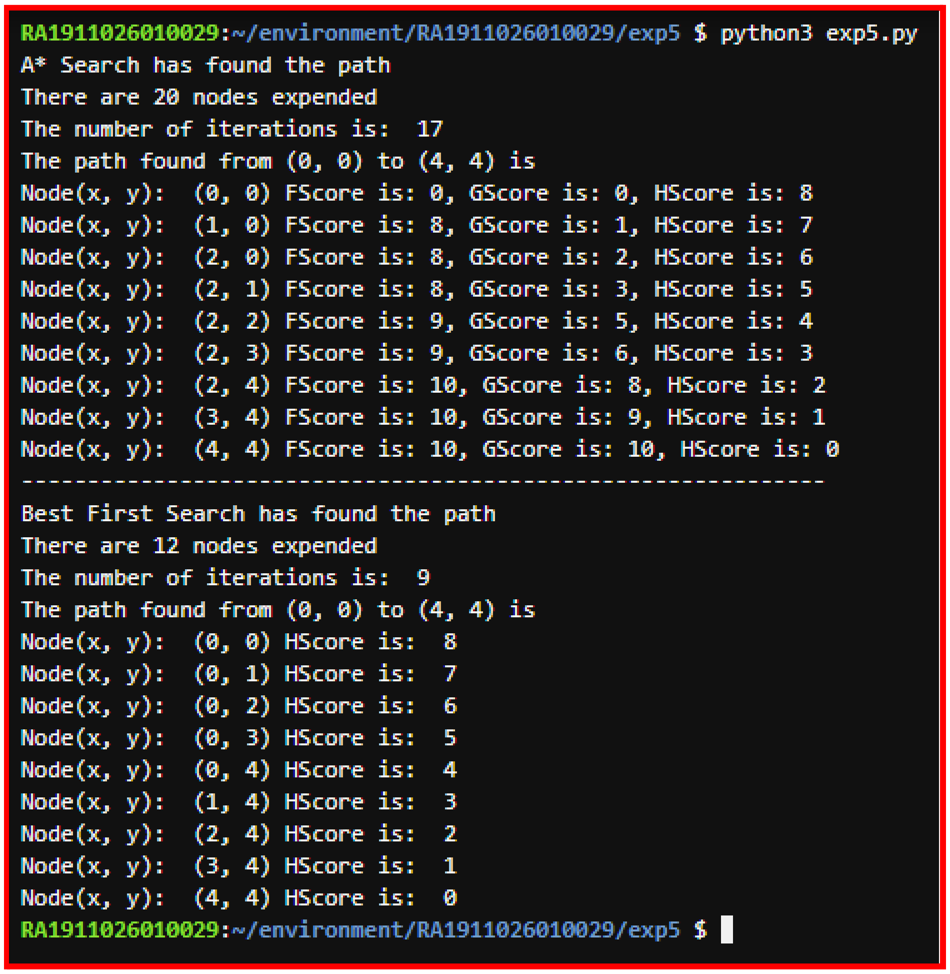
## Input

Size of grid, start position and goal position, value



```
5
[0,0]  [4,4]
[[1 1 1 1 2]
 [1 6 1 6 1]
 [1 6 2 2 3]
 [1 6 2 6 1]
 [3 1 1 1 1]]
```

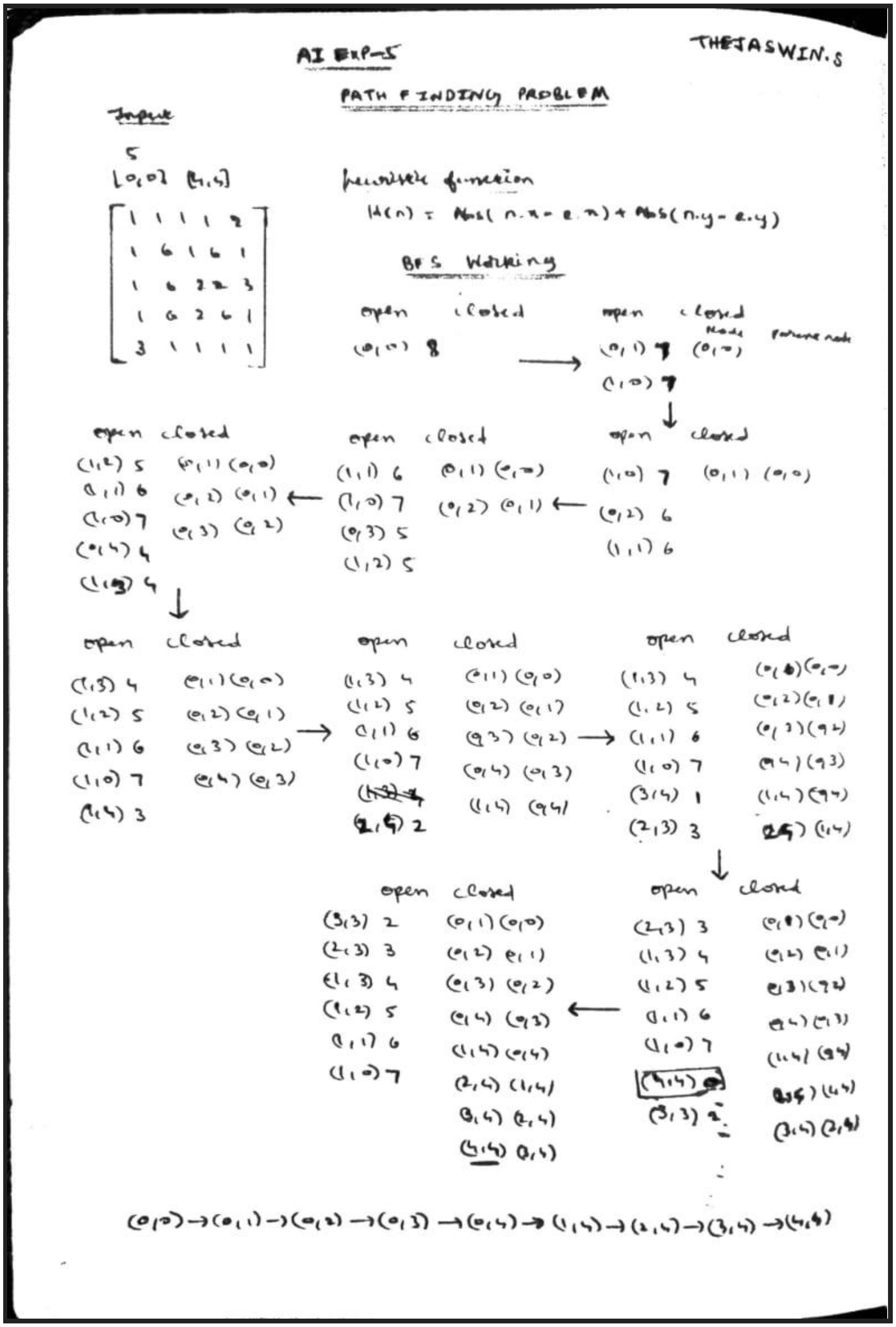
**Output**
The output prints the final path, number of nodes expanded during search, number of iterations used, and the information of each node in the path for the result found by A* and BFS. If A*/ BFS cannot find the path, program will print no path found message.



```
RA1911026010029:~/environment/RA1911026010029/exp5 $ python3 exp5.py
A* Search has found the path
There are 20 nodes expended
The number of iterations is:  17
The path found from (0, 0) to (4, 4) is
Node(x, y):  (0, 0) FScore is: 0, GScore is: 0, HScore is: 8
Node(x, y):  (1, 0) FScore is: 8, GScore is: 1, HScore is: 7
Node(x, y):  (2, 0) FScore is: 8, GScore is: 2, HScore is: 6
Node(x, y):  (2, 1) FScore is: 8, GScore is: 3, HScore is: 5
Node(x, y):  (2, 2) FScore is: 9, GScore is: 5, HScore is: 4
Node(x, y):  (2, 3) FScore is: 9, GScore is: 6, HScore is: 3
Node(x, y):  (2, 4) FScore is: 10, GScore is: 8, HScore is: 2
Node(x, y):  (3, 4) FScore is: 10, GScore is: 9, HScore is: 1
Node(x, y):  (4, 4) FScore is: 10, GScore is: 10, HScore is: 0
-------------------------------------------------------------
Best First Search has found the path
There are 12 nodes expended
The number of iterations is:  9
The path found from (0, 0) to (4, 4) is
Node(x, y):  (0, 0) HScore is:  8
Node(x, y):  (0, 1) HScore is:  7
Node(x, y):  (0, 2) HScore is:  6
Node(x, y):  (0, 3) HScore is:  5
Node(x, y):  (0, 4) HScore is:  4
Node(x, y):  (1, 4) HScore is:  3
Node(x, y):  (2, 4) HScore is:  2
Node(x, y):  (3, 4) HScore is:  1
Node(x, y):  (4, 4) HScore is:  0
RA1911026010029:~/environment/RA1911026010029/exp5 $
```

**Verification:**

RA1911026010029
Thejaswin S



AI EXP-5 THEJASWIN.S

PATH FINDING PROBLEM

Input

5
[0,0] [4,4]

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 2 \\ 1 & 6 & 1 & 6 & 1 \\ 1 & 6 & 2 & 2 & 3 \\ 1 & 6 & 2 & 6 & 1 \\ 3 & 1 & 1 & 1 & 1 \end{bmatrix}$$

heuristic function

H(n) = Abs( n.x - e.x) + Abs(n.y - e.y)

BFS Working

| open | closed |
|---|---|
| (0,0) 8 | |

$\longrightarrow$

| open | closed Node | Parent node |
|---|---|---|
| (0,1) 7 | (0,0) | |
| (1,0) 7 | | |

$\downarrow$

| open | closed | |
|---|---|---|
| (1,0) 7 | (0,1) | (0,0) |
| (0,2) 6 | | |
| (1,1) 6 | | |

$\longleftarrow$

| open | closed | |
|---|---|---|
| (1,1) 6 | (0,1) | (0,0) |
| (1,0) 7 | (0,2) | (0,1) |
| (0,3) 5 | | |
| (1,2) 5 | | |

$\longleftarrow$

| open | closed | |
|---|---|---|
| (1,2) 5 | (0,1) | (0,0) |
| (1,1) 6 | (0,2) | (0,1) |
| (1,0) 7 | (0,3) | (0,2) |
| (0,4) 4 | | |
| (1,3) 4 | | |

$\downarrow$

| open | closed | |
|---|---|---|
| (1,3) 4 | (0,1) | (0,0) |
| (1,2) 5 | (0,2) | (0,1) |
| (1,1) 6 | (0,3) | (0,2) |
| (1,0) 7 | (0,4) | (0,3) |
| (1,4) 3 | | |

$\longrightarrow$

| open | closed | |
|---|---|---|
| (1,3) 4 | (0,1) | (0,0) |
| (1,2) 5 | (0,2) | (0,1) |
| (1,1) 6 | (0,3) | (0,2) |
| (1,0) 7 | (0,4) | (0,3) |
| ~~(1,3) 4~~ | (1,4) | (0,4) |
| (2,4) 2 | | |

$\longrightarrow$

| open | closed | |
|---|---|---|
| (1,3) 4 | (0,1) | (0,0) |
| (1,2) 5 | (0,2) | (0,1) |
| (1,1) 6 | (0,3) | (0,2) |
| (1,0) 7 | (0,4) | (0,3) |
| (3,4) 1 | (1,4) | (0,4) |
| (2,3) 3 | (2,4) | (1,4) |

$\downarrow$

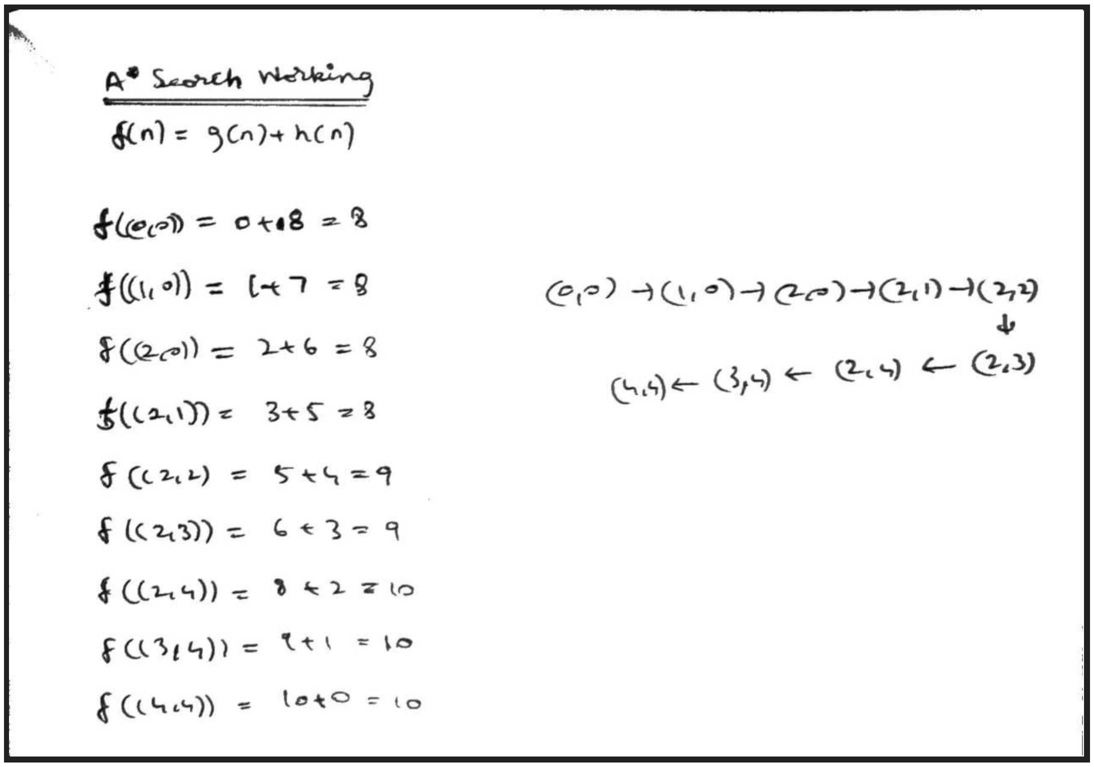
| open | closed | |
|---|---|---|
| (2,3) 3 | (0,1) | (0,0) |
| (1,3) 4 | (0,2) | (0,1) |
| (1,2) 5 | (0,3) | (0,2) |
| (1,1) 6 | (0,4) | (0,3) |
| (1,0) 7 | (1,4) | (0,4) |
| $\boxed{(4,4)\ 0}$ | (2,4) | (1,4) |
| (3,3) 2 | (3,4) | (2,4) |

$\longleftarrow$

| open | closed | |
|---|---|---|
| (3,3) 2 | (0,1) | (0,0) |
| (2,3) 3 | (0,2) | (0,1) |
| (1,3) 4 | (0,3) | (0,2) |
| (1,2) 5 | (0,4) | (0,3) |
| (1,1) 6 | (1,4) | (0,4) |
| (1,0) 7 | (2,4) | (1,4) |
| | (3,4) | (2,4) |
| | $\underline{(4,4)}$ | (3,4) |

(0,0) → (0,1) → (0,2) → (0,3) → (0,4) → (1,4) → (2,4) → (3,4) → (4,4)

RA1911026010029
Thejaswin S



**A* Search Working**

$f(n) = g(n) + h(n)$

$f((0,0)) = 0+8 = 8$

$f((1,0)) = 1+7 = 8$

$f((2,0)) = 2+6 = 8$

$f((2,1)) = 3+5 = 8$

$f((2,2)) = 5+4 = 9$

$f((2,3)) = 6+3 = 9$

$f((2,4)) = 8+2 = 10$

$f((3,4)) = 9+1 = 10$

$f((4,4)) = 10+0 = 10$

$$\begin{tikzcd}
 & (0,0) \arrow[r] & (1,0) \arrow[r] & (2,0) \arrow[r] & (2,1) \arrow[r] & (2,2) \arrow[d] \\
 & (4,4) & (3,4) \arrow[l] & (2,4) \arrow[l] & & (2,3) \arrow[ll]
\end{tikzcd}$$

**Observation:**
Thus the concept of Best First Search and A* Search algorithm is well studied with the example of real-life scenario example implementation. I observed that Best First Search Algorithm performed well as compared to the A* algorithm in finding paths in the grid/Maze.

**Result:**
The program for pathfinding in the maze using A* and Best First Search algorithm has been successfully implemented in Python.