# Exercise 1: Implement Informed and Uninformed Search

## **1.A) BFS**

### **Tool**: Python

### **Libraries Used**: queue

### **Sample Problem**: Given a binary maze with obstacles and traversable blocks, find the shortest path between a source cell and destination cell. We can move up, down, left and right from a row-column pair

### **Input Type**: Binary matrix, source indices, destination indices

Input Binary Matrix:

inputMaze = [[1, 0, 0, 0],  
 [1, 1, 0, 1],  
 [0, 1, 0, 0],  
 [1, 1, 1, 1]]

src = [0, 0]

dest = [3, 3]

### **Logic / Search Technique**: Breadth First Search

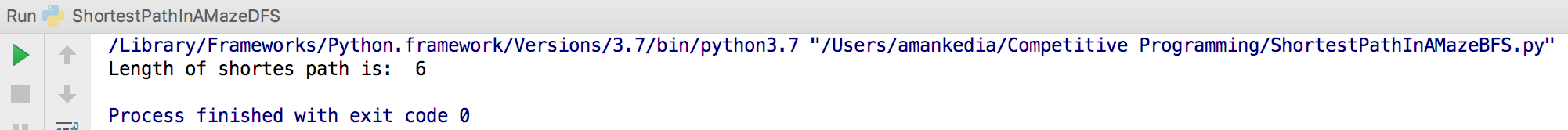
### **Output Type**: Integer representing shortest path

### **Implementation**

**import** queue  
  
*#These two lists help in getting coordinates of the adjacent 4 cells*rowNums = [0, -1, 0, 1]  
colNums = [-1, 0, 1, 0]  
  
**class** Point():  
  
 *# Objects of this class would be used to define coordinates in the matrix* **def** \_\_init\_\_(self, x, y):  
 self.x = x  
 self.y = y  
  
**class** Node():  
  
 *#Objects of this class would be used to define every entry in the matrix  
 #including its coordinates and distance from source* **def** \_\_init\_\_(self, coordinates, dist):  
 self.coordinates = coordinates  
 self.dist = dist  
  
**def** isSafe(x, y, nrow, ncol):  
  
 *#This method would help to evaluate if a pair of coordinates is valid or not* **if** x >= 0 **and** x < nrow **and** y >= 0 **and** y < ncol:  
 **return True  
 return False  
  
  
def** shortestPathInAMaze(maze, src, dest):  
  
 nrow = len(maze)  
 ncol = len(maze[0])  
  
 val = BFS(maze, src, dest, nrow, ncol)  
  
 **return** val  
  
  
**def** BFS(maze, src, dest, nrow, ncol):  
  
 visited = []  
  
 *# if src or destination is an obstacle, we cannot have a path* **if** maze[src.x][src.y] **is not** 1 **or** maze[dest.x][dest.y] **is not** 1:  
 **return** -1  
  
 **for** i **in** range(len(maze)):  
 visited.append([**False**] \* len(maze[i]))  
  
 *#Mark the source as visited* visited[src.x][src.y] = **True** q = queue.Queue(maxsize = (nrow\*ncol))  
  
 *#Add source to queue* q.put(Node(src, 0))  
  
 **while not** q.empty():  
 current = q.get()  
  
 point = current.coordinates  
  
 *#If coordinates of cuurent node are same as destination, the goal has been reached* **if** point.x == dest.x **and** point.y == dest.y:  
 **return** current.dist  
  
 **for** i **in** range(0, 4):  
 row = point.x + rowNums[i]  
 col = point.y + colNums[i]  
  
 *#add the adjacent node to queue if it is a valid coordinate, it is not an obstacle and has not been visited yet* **if** isSafe(row, col, nrow, ncol) **and** maze[row][col] **is not** 0 **and** visited[row][col] **is False**:  
 visited[row][col] = **True** newNode = Node(Point(row, col), current.dist + 1)  
 q.put(newNode)  
  
 *#If a path has not been found then return -1* **return** -1  
  
  
**if** \_\_name\_\_ == **"\_\_main\_\_"**:  
  
 inputMaze = [[1, 0, 0, 0]  
 [1, 1, 0, 1],  
 [0, 1, 0, 0],  
 [1, 1, 1, 1]]  
  
 src = [0, 0]  
 dest = [3, 3]  
  
 srcObject = Point(src[0], src[1])  
 destObject = Point(dest[0], dest[1])  
  
 val = shortestPathInAMaze(inputMaze, srcObject, destObject)  
  
 **if** val == -1:  
 print(**"Path does not exist"**)  
 **else**:  
 print(**"Length of shortes path is: "**, val)

### **Output**: 6

### **Screenshot**

****

## **1.B) A\* algorithm**

### **Tool**: Python

### **Libraries Used**: math

### **Sample Problem**:

Find the shortest distance between a pair of points such that one can move east, west, north, south, south-east, north-east, south-west and north-west. Use Euclidean distance as the for calculating the heuristic function.

### **Input Type**: Binary matrix, source indices, destination indices

maze = [[1, 1, 1, 1, 0, 1, 1, 1, 1, 1],  
 [1, 1, 1, 1, 0, 1, 1, 1, 1, 1],  
 [1, 1, 1, 1, 0, 1, 1, 1, 1, 1],  
 [1, 1, 1, 1, 0, 1, 1, 1, 1, 1],  
 [1, 1, 1, 1, 0, 1, 1, 1, 1, 1],  
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],  
 [1, 1, 1, 1, 0, 1, 1, 1, 1, 1],  
 [1, 1, 1, 1, 0, 1, 1, 1, 1, 1],  
 [1, 1, 1, 1, 0, 1, 1, 1, 1, 1],  
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]]  
  
start = (0, 0)  
end = (7, 6)

### **Logic / Search Technique:** A\* Search

### **Output Type**: Set of Pairs of Integer representing shortest path/route

### **Implementation**

**import** math  
  
**class** Node():  
 *"""A node class for A\* Pathfinding"""* **def** \_\_init\_\_(self, parent=**None**, position=**None**):  
 self.parent = parent  
 self.position = position  
  
 self.g = 0  
 self.h = 0  
 self.f = 0  
  
 **def** \_\_eq\_\_(self, other):  
 **return** self.position == other.position  
  
  
**def** astar(maze, start, end):  
 *"""Returns a list of tuples as a path from the given start to the given end in the given maze"""  
  
 # Create start and end node* start\_node = Node(**None**, start)  
 start\_node.g = start\_node.h = start\_node.f = 0  
 end\_node = Node(**None**, end)  
 end\_node.g = end\_node.h = end\_node.f = 0  
  
 *# Initialize both open and closed list* open\_list = []  
 closed\_list = []  
  
 *# Add the start node* open\_list.append(start\_node)  
  
 *# Loop until you find the end* **while** len(open\_list) > 0:  
  
 *# Get the current node* current\_node = open\_list[0]  
 current\_index = 0  
 **for** index, item **in** enumerate(open\_list):  
 **if** item.f < current\_node.f:  
 current\_node = item  
 current\_index = index  
  
 *# Pop current off open list, add to closed list* open\_list.pop(current\_index)  
 closed\_list.append(current\_node)  
  
 *# Found the goal* **if** current\_node == end\_node:  
 path = []  
 current = current\_node  
 **while** current **is not None**:  
 path.append(current.position)  
 current = current.parent  
 **return** path[::-1] *# Return reversed path  
  
 # Generate children* children = []  
 **for** new\_position **in** [(0, -1), (0, 1), (-1, 0), (1, 0), (-1, -1), (-1, 1), (1, -1), (1, 1)]: *# Adjacent squares  
  
 # Get node position* node\_position = (current\_node.position[0] + new\_position[0], current\_node.position[1] + new\_position[1])  
  
 *# Make sure within range* **if** node\_position[0] > (len(maze) - 1) **or** node\_position[0] < 0 **or** node\_position[1] > (len(maze[len(maze)-1]) -1) **or** node\_position[1] < 0:  
 **continue** *# Make sure walkable terrain* **if** maze[node\_position[0]][node\_position[1]] != 1:  
 **continue** *# Create new node* new\_node = Node(current\_node, node\_position)  
  
 *# Append* children.append(new\_node)  
  
 *# Loop through children* **for** child **in** children:  
  
 *# Child is on the closed list* **for** closed\_child **in** closed\_list:  
 **if** child == closed\_child:  
 **continue** *# Create the f, g, and h values* child.g = current\_node.g + 1  
 child.h = math.sqrt(((child.position[0] - end\_node.position[0]) \*\* 2) + ((child.position[1] - end\_node.position[1]) \*\* 2))  
 child.f = child.g + child.h  
  
 *# Child is already in the open list* **for** open\_node **in** open\_list:  
 **if** child == open\_node **and** child.g > open\_node.g:  
 **continue** *# Add the child to the open list* open\_list.append(child)  
  
  
**if** \_\_name\_\_ == **'\_\_main\_\_'**:  
 maze = [[1, 1, 1, 1, 0, 1, 1, 1, 1, 1],  
 [1, 1, 1, 1, 0, 1, 1, 1, 1, 1],  
 [1, 1, 1, 1, 0, 1, 1, 1, 1, 1],  
 [1, 1, 1, 1, 0, 1, 1, 1, 1, 1],  
 [1, 1, 1, 1, 0, 1, 1, 1, 1, 1],  
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],  
 [1, 1, 1, 1, 0, 1, 1, 1, 1, 1],  
 [1, 1, 1, 1, 0, 1, 1, 1, 1, 1],  
 [1, 1, 1, 1, 0, 1, 1, 1, 1, 1],  
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]]  
  
 start = (0, 0)  
 end = (7, 6)  
  
 path = astar(maze, start, end)  
 print(path)

### **Output**: [(0, 0), (1, 1), (2, 2), (3, 3), (4, 3), (5, 4), (6, 5), (7, 6)]

### **Screenshot**

