

**SCHOOL OF INFORMATION TECHNOLOGY AND ENGINEERING**

**DIGITAL ASSIGNMENT- 1**

**WINTER SEMESTER 2023-24**

**COURSE: ARTIFICIAL INTELLIGENCE LAB**

**COURSE CODE: BITE308P**

**NAME: POLI VARDHINI REDDY**

**REGISTER NUMBER: 21BIT0382**

1. **A\* ALGORITHM**

**CODE:**

class Node():

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):

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

open\_list = []

closed\_list = []

open\_list.append(start\_node)

while len(open\_list) > 0:

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

open\_list.pop(current\_index)

closed\_list.append(current\_node)

if current\_node == end\_node:

path = []

current = current\_node

while current is not None:

path.append(current.position)

current = current.parent

return path[::-1]

children = []

for new\_position in [(0, -1), (0, 1), (-1, 0), (1, 0), (-1, -1), (-1, 1), (1, -1), (1, 1)]:

node\_position = (current\_node.position[0] + new\_position[0], current\_node.position[1] + new\_position[1])

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

if maze[node\_position[0]][node\_position[1]] != 0:

continue

new\_node = Node(current\_node, node\_position)

children.append(new\_node)

for child in children:

for closed\_child in closed\_list:

if child == closed\_child:

continue

child.g = current\_node.g + 1

child.h = ((child.position[0] - end\_node.position[0]) \*\* 2) + ((child.position[1] - end\_node.position[1]) \*\* 2)

child.f = child.g + child.h

for open\_node in open\_list:

if child == open\_node and child.g > open\_node.g:

continue

open\_list.append(child)

def main():

maze = [[0, 0, 0, 0, 1, 0, 0, 0, 0, 0],

[0, 0, 0, 0, 1, 0, 0, 0, 0, 0],

[0, 0, 0, 0, 1, 0, 0, 0, 0, 0],

[0, 0, 0, 0, 1, 0, 0, 0, 0, 0],

[0, 0, 0, 0, 1, 0, 0, 0, 0, 0],

[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],

[0, 0, 0, 0, 1, 0, 0, 0, 0, 0],

[0, 0, 0, 0, 1, 0, 0, 0, 0, 0],

[0, 0, 0, 0, 1, 0, 0, 0, 0, 0],

[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]

start = (0, 0)

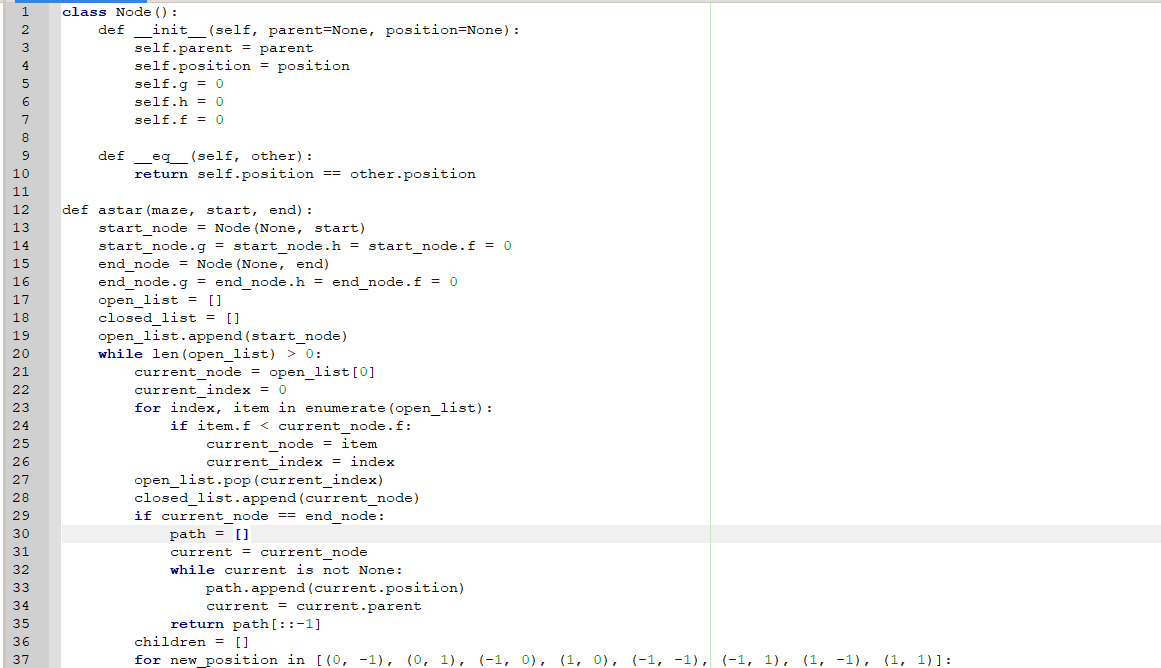
end = (7, 6)

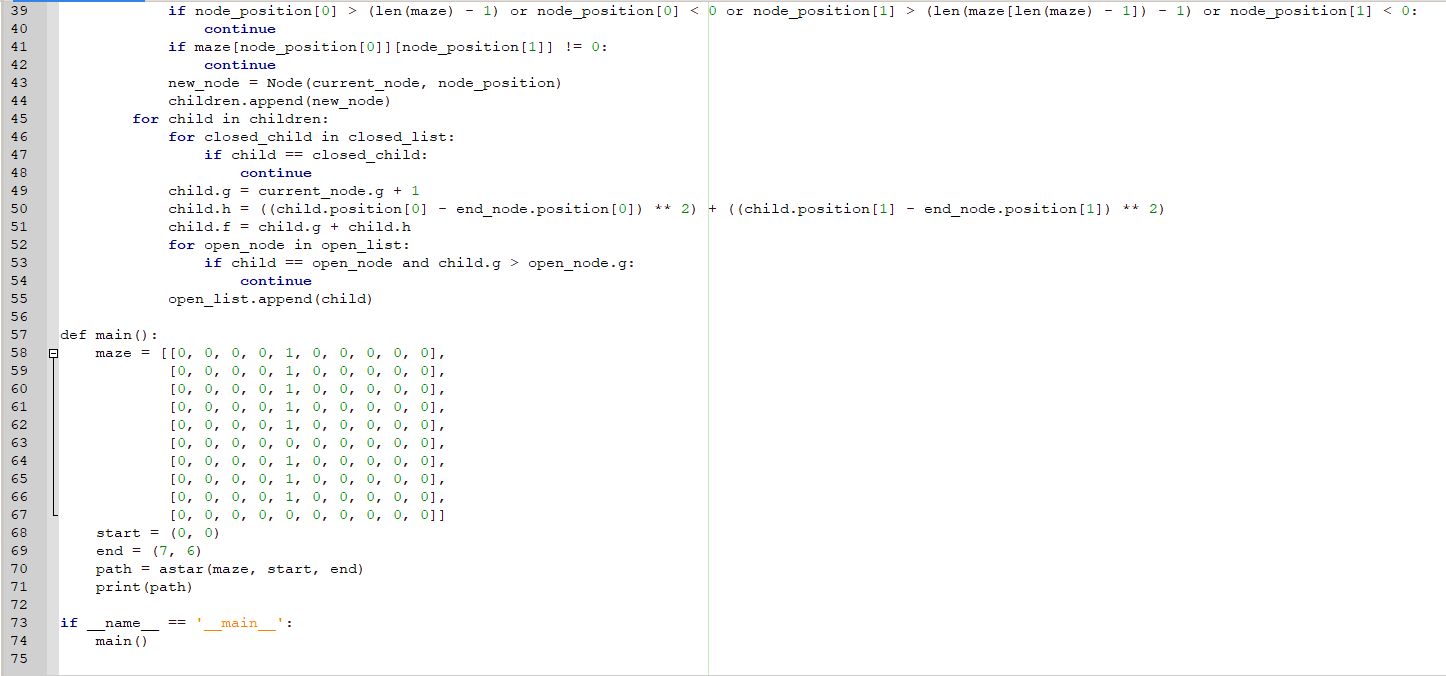
path = astar(maze, start, end)

print(path)

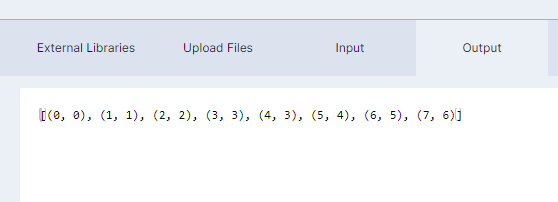
if \_\_name\_\_ == '\_\_main\_\_':

main()





**OUTPUT-**



1. **DFS ALGORTIHM**

**CODE:**

from \_\_future\_\_ import print\_function

def dfs(graph, start, visited=None):

if visited is None:

visited = set()

visited.add(start)

print(start),

for next\_node in graph[start] - visited:

dfs(graph, next\_node, visited)

return visited

# Example graph represented as an adjacency list

graph = {

'A': {'B', 'C'},

'B': {'A', 'D', 'E'},

'C': {'A', 'F'},

'D': {'B'},

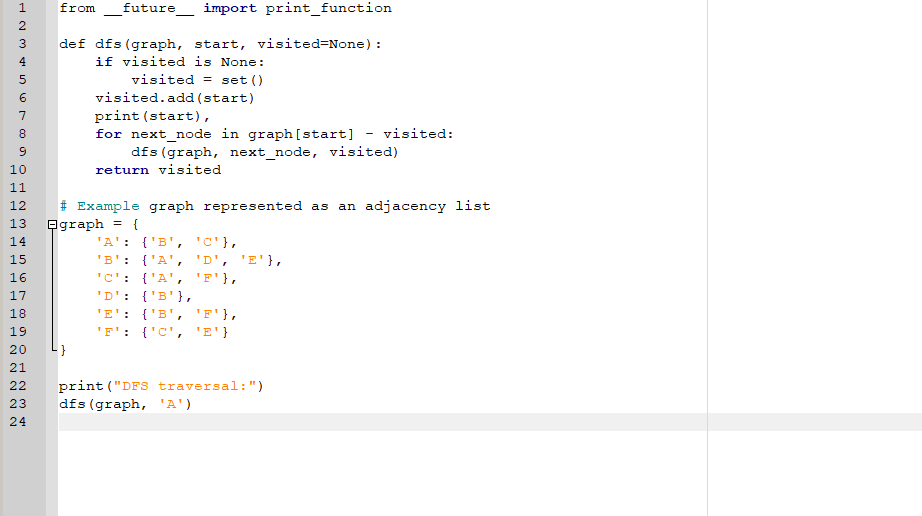
'E': {'B', 'F'},

'F': {'C', 'E'}

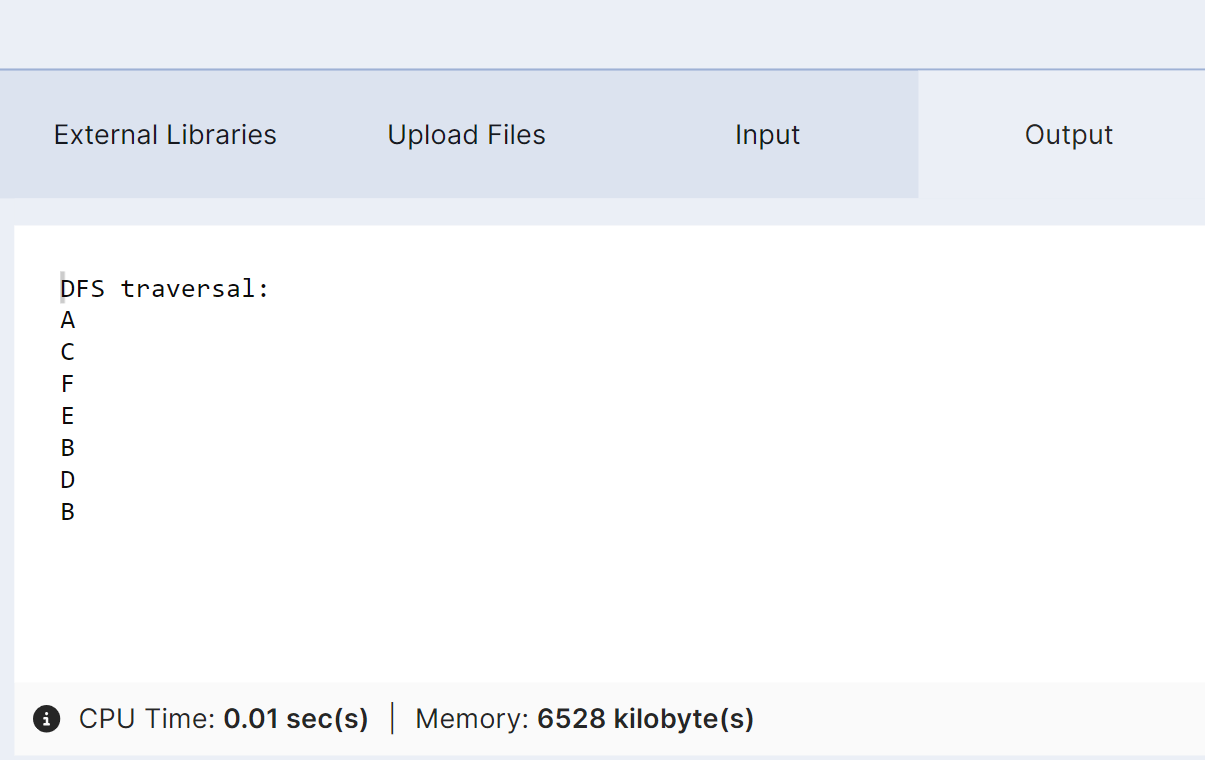
}

print("DFS traversal:")

dfs(graph, 'A')



**OUTPUT:**



1. **BFS ALGORITHM**

**CODE:**

from collections import deque

def bfs(graph, start):

visited = set()

queue = deque([start])

visited.add(start)

while queue:

vertex = queue.popleft()

print vertex,

for neighbor in graph[vertex]:

if neighbor not in visited:

queue.append(neighbor)

visited.add(neighbor)

# Example graph represented as an adjacency list

graph = {

'A': ['B', 'C'],

'B': ['A', 'D', 'E'],

'C': ['A', 'F'],

'D': ['B'],

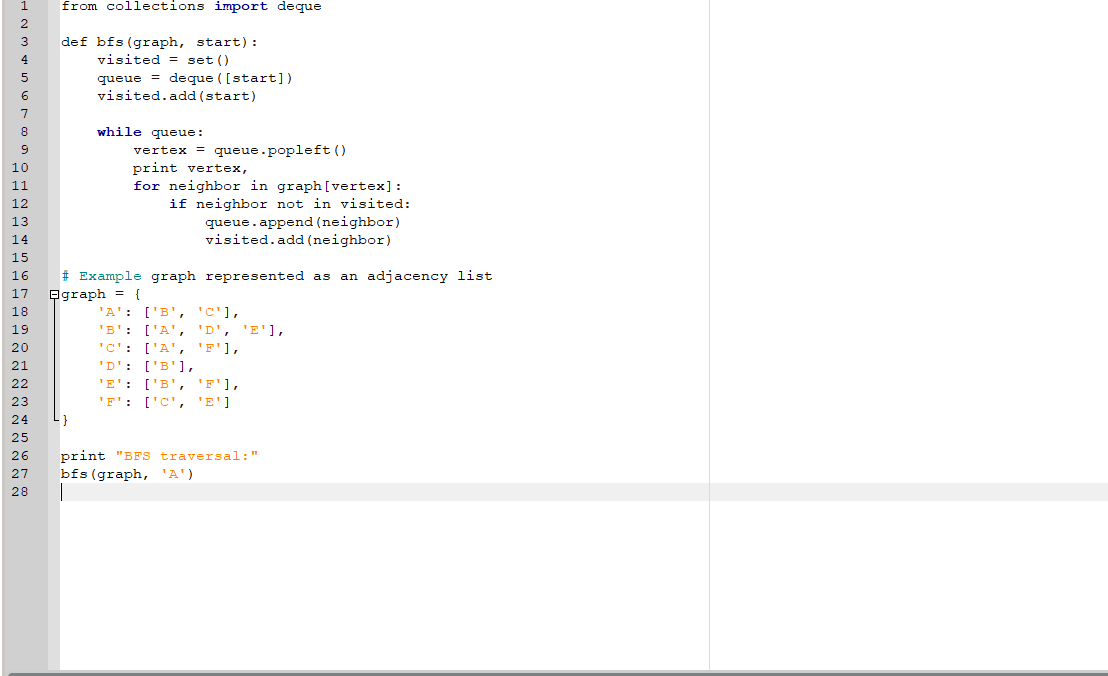
'E': ['B', 'F'],

'F': ['C', 'E']

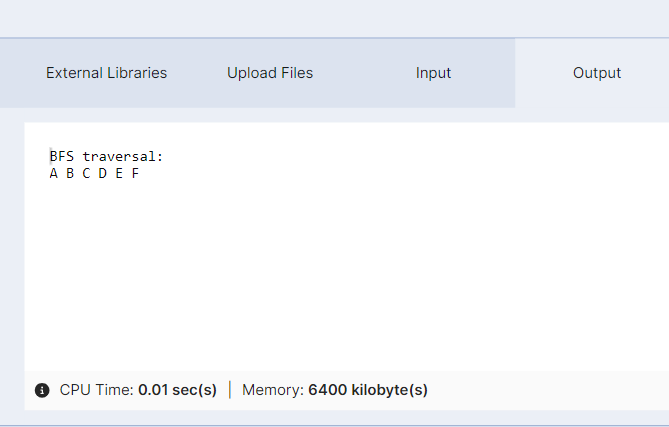
}

print "BFS traversal:"

bfs(graph, 'A')



**OUTPUT:**



1. **MINIMAX ALGORITHM**

**CODE:**

def minimax(position, depth, maximizing\_player):

if depth == 0 or game\_over(position):

return evaluate\_position(position)

if maximizing\_player:

max\_eval = float('-inf')

for move in possible\_moves(position):

eval = minimax(move, depth - 1, False)

max\_eval = max(max\_eval, eval)

return max\_eval

else:

min\_eval = float('inf')

for move in possible\_moves(position):

eval = minimax(move, depth - 1, True)

min\_eval = min(min\_eval, eval)

return min\_eval

def game\_over(position):

return position == "Terminal"

def evaluate\_position(position):

return 1

def possible\_moves(position):

return ["Move 1", "Move 2", "Move 3"]

initial\_position = "Initial Position"

best\_move = None

best\_score = float('-inf')

for move in possible\_moves(initial\_position):

score = minimax(move, depth=3, maximizing\_player=False)

if score > best\_score:

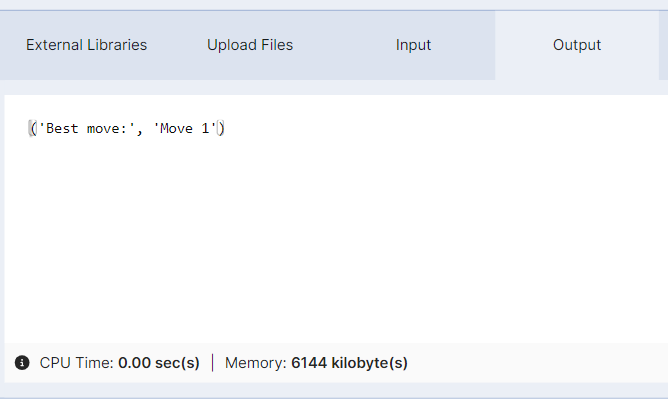
best\_score = score

best\_move = move

print("Best move:", best\_move)



**OUTPUT:**



1. **ALPHABETA PRUNING ALGORITHM**

**CODE:**

def alphabeta(position, depth, alpha, beta, maximizing\_player):

if depth == 0 or game\_over(position):

return evaluate\_position(position)

if maximizing\_player:

max\_eval = float('-inf')

for move in possible\_moves(position):

eval = alphabeta(move, depth - 1, alpha, beta, False)

max\_eval = max(max\_eval, eval)

alpha = max(alpha, eval)

if beta <= alpha:

break

return max\_eval

else:

min\_eval = float('inf')

for move in possible\_moves(position):

eval = alphabeta(move, depth - 1, alpha, beta, True)

min\_eval = min(min\_eval, eval)

beta = min(beta, eval)

if beta <= alpha:

break

return min\_eval

def game\_over(position):

return position == "Terminal"

def evaluate\_position(position):

if position == "Move 3":

return 100 # Assign a high score to "Move 3"

else:

return 1

def possible\_moves(position):

return ["Move 1", "Move 2", "Move 3"] # Ensure "Move 3" is one of the options

initial\_position = "Initial Position"

best\_move = None

alpha = float('-inf')

beta = float('inf')

for move in possible\_moves(initial\_position):

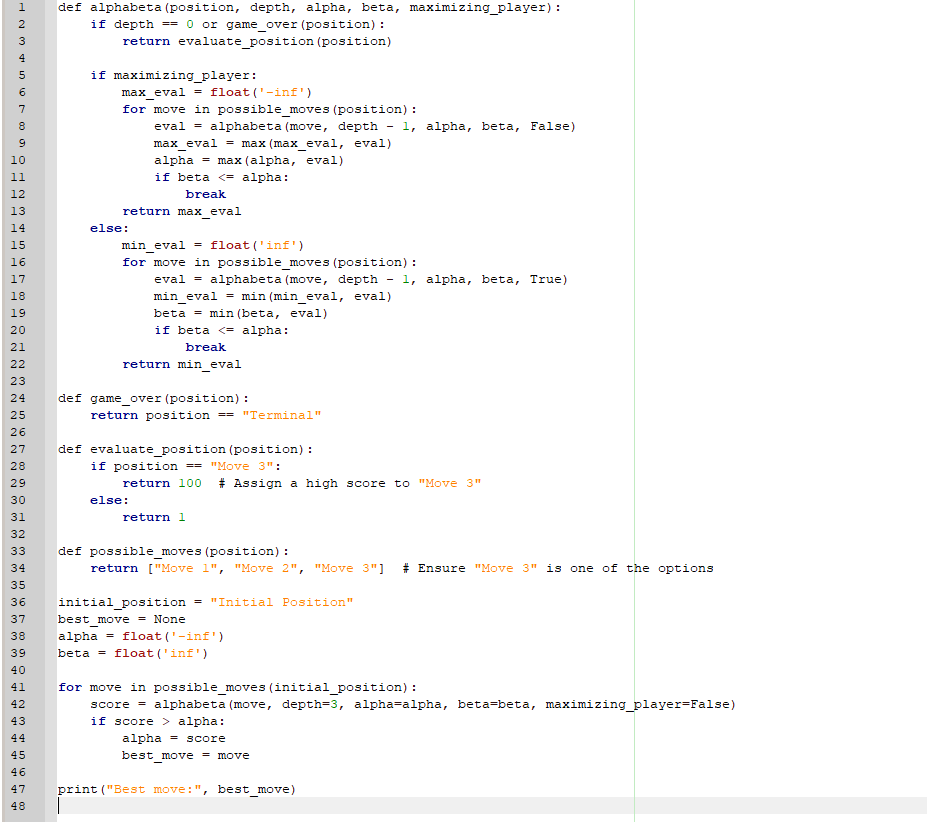
score = alphabeta(move, depth=3, alpha=alpha, beta=beta, maximizing\_player=False)

if score > alpha:

alpha = score

best\_move = move

print("Best move:", best\_move)



**OUTPUT:**

