

**CENTRAL UNIVERSITY OF RAJASTHAN**

**Department of Computer Science**

**NAME:** AGHIL.M

**ENROLLMENT NO. :**2024MSCS021

**PROGRAMME NAME:** M.Sc. Computer Science.

**COURSE NAME:** Artificial intelligence Lab

**COURSE CODE:** CSC-407

**SUBMITTED TO: Dr. Gaurav Meena**

**Program: - 01**

**Objective: -**

**The objective of the Breadth-First Search (BFS) algorithm is to traverse or search through a graph or tree systematically to explore all reachable nodes, layer by layer, from a starting node. Depending on the context, BFS can achieve various goals:**

**Code: -**

from collections import deque

def bfs(graph, start):

    visited = set()

    queue = deque([start])

    visited.add(start)

    print("BFS Traversal:", end=' ')

    while queue:

        node = queue.popleft()

        print(node, end=' ')

        for neighbor in graph[node]:

            if neighbor not in visited:

                visited.add(neighbor)

                queue.append(neighbor)

graph = {

    'A': ['B', 'C'],

    'B': ['A', 'D', 'E'],

    'C': ['A', 'F'],

    'D': ['B'],

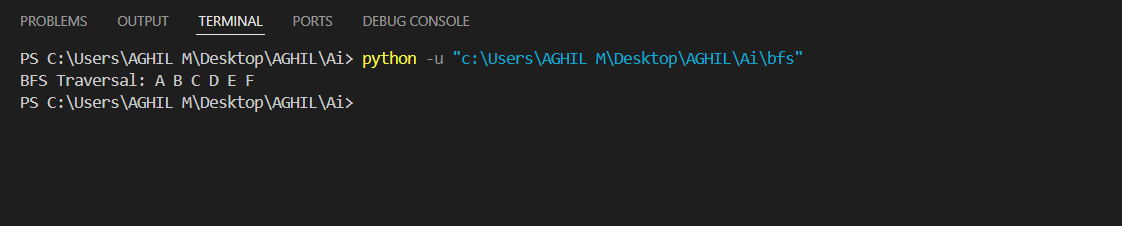
    'E': ['B', 'F'],

    'F': ['C', 'E']

}

bfs(graph, 'A')

**Output**



**Time Complexity: -**

Best case: - O(V+E)

Average case: - O(V+E)

Worst case: - O(V+E)

**Program: - 02**

**Objective: -**

**The objective of the Depth-First Search (DFS) algorithm is to explore all the nodes in a graph or tree systematically by going as deep as possible along each branch before backtracking. It is used for traversal and search operations in graphs and trees.**

**Code: -**

from collections import defaultdict

class Graph:

    def \_\_init\_\_(self):

        self.graph = defaultdict(list)

    def addEdge(self, u, v):

        self.graph[u].append(v)

    def DFSUtil(self, v, visited):

        visited.add(v)

        print(v, end=' ')

        for neighbour in self.graph[v]:

            if neighbour not in visited:

                self.DFSUtil(neighbour, visited)

    def DFS(self, v):

        visited = set()

        self.DFSUtil(v, visited)

if \_\_name\_\_ == "\_\_main\_\_":

    g = Graph()

    g.addEdge(0, 1)

    g.addEdge(0, 2)

    g.addEdge(1, 2)

    g.addEdge(2, 0)

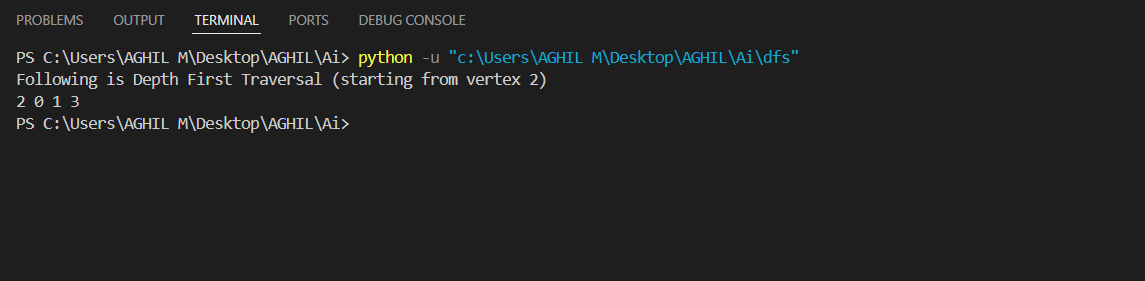
    g.addEdge(2, 3)

    g.addEdge(3, 3)

    print("Following is Depth First Traversal (starting from vertex 2)")

    g.DFS(2)

**Output**

****

**Time Complexity: -**

**Best case: - O(V)**

**Average case: - O(V+E)**

**Worst case: - O(V+E)**

**Program: - 03**

**Objective: -**

**The objective of the A\* algorithm is to find the optimal (shortest or least-cost) path from a starting node to a target node in a graph or grid while minimizing computational overhead.**

**Code: -**

import heapq

def a\_star\_search(graph, start, goal, heuristic):

    open\_set = [(0 + heuristic(start, goal), start, [start])]

    closed\_set = set()

    while open\_set:

        f\_score, current, path = heapq.heappop(open\_set)

        if current == goal:

            return path

        closed\_set.add(current)

        for neighbor, cost in graph[current].items():

            if neighbor in closed\_set:

                continue

            g\_score = f\_score - heuristic(current, goal) + cost

            h\_score = heuristic(neighbor, goal)

            total\_score = g\_score + h\_score

            heapq.heappush(open\_set, (total\_score, neighbor, path + [neighbor]))

    return None

graph = {

    1: {2: 2, 3: 4},

    2: {1: 2, 4: 1},

    3: {1: 4, 5: 3},

    4: {2: 1, 6: 5},

    5: {3: 3, 6: 2},

    6: {4: 5, 5: 2}

}

def manhattan\_distance(node, goal):

    return abs(node - goal)

start\_node = 1

goal\_node = 6

path = a\_star\_search(graph, start\_node, goal\_node, manhattan\_distance)

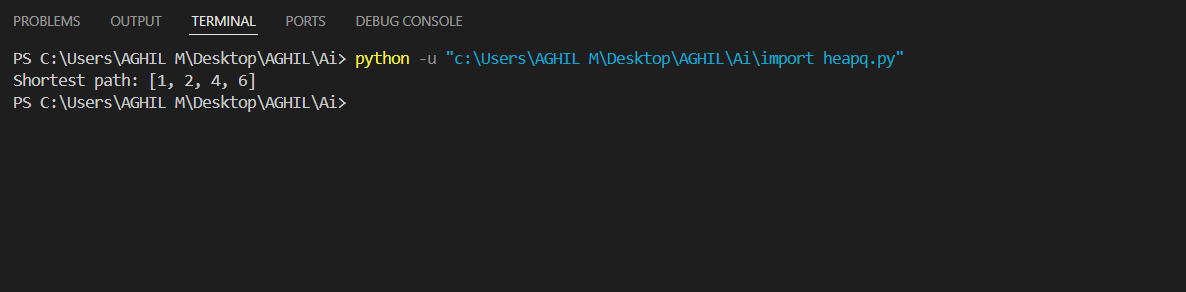
if path:

    print("Shortest path:", path)

else:

    print("No path found.")

**Output**

****

**Time Complexity: -**

Best case: - **O(V)**

Average case: - **O(V+E)**

Worst case: - **O(V+E)**

**Program: - 04**

**Objective: -**

**The AO\* (And-Or Search) algorithm is an extension of the A\* algorithm used primarily for solving problems involving AND/OR search spaces, which are often encountered in scenarios like planning, game trees, and multi-agent systems. The objective of AO\* is to find an optimal solution in search spaces that involve both AND and OR relations between subgoals and states.**

**Code: -**

import heapq

class Node:

    def \_\_init\_\_(self, position, parent=None):

        self.position = position

        self.parent = parent

        self.g = 0  # Cost from start to current node

        self.h = 0  # Heuristic cost estimate to goal

        self.f = 0  # Total cost (g + h)

    def \_\_eq\_\_(self, other):

        return self.position == other.position

    def \_\_lt\_\_(self, other):

        return self.f < other.f

def heuristic(a, b):

    return abs(a[0] - b[0]) + abs(a[1] - b[1])

def a\_star(graph, start, end):

    open\_list = []

    closed\_list = set()

    start\_node = Node(start)

    end\_node = Node(end)

    heapq.heappush(open\_list, start\_node)

    while open\_list:

        current\_node = heapq.heappop(open\_list)

        closed\_list.add(current\_node.position)

        if current\_node == end\_node:

            path = []

            while current\_node:

                path.append(current\_node.position)

                current\_node = current\_node.parent

            return path[::-1]

        neighbors = [(0, -1), (0, 1), (-1, 0), (1, 0)]

        for offset in neighbors:

            neighbor\_position = (current\_node.position[0] + offset[0], current\_node.position[1] + offset[1])

            if neighbor\_position not in graph or neighbor\_position in closed\_list:

                continue

            neighbor\_node = Node(neighbor\_position, current\_node)

            neighbor\_node.g = current\_node.g + 1

            neighbor\_node.h = heuristic(neighbor\_node.position, end\_node.position)

            neighbor\_node.f = neighbor\_node.g + neighbor\_node.h

            if any(open\_node for open\_node in open\_list if neighbor\_node == open\_node and neighbor\_node.g > open\_node.g):

                continue

            heapq.heappush(open\_list, neighbor\_node)

    return None

# Example usage

graph = {

    (0, 0): 1, (0, 1): 1, (0, 2): 1,

    (1, 0): 1, (1, 1): 1, (1, 2): 1,

    (2, 0): 1, (2, 1): 1, (2, 2): 1

}

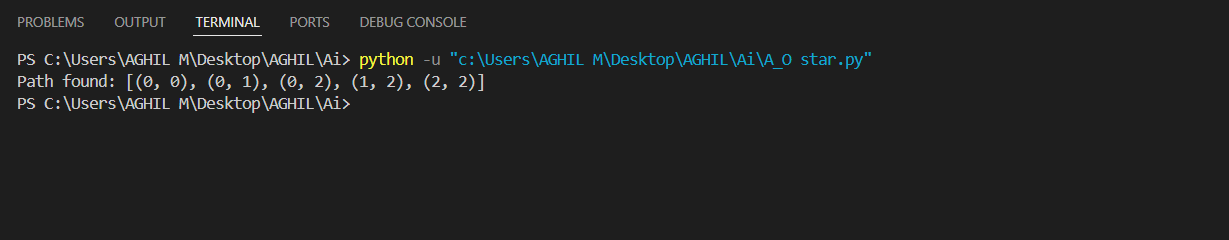
start = (0, 0)

end = (2, 2)

path = a\_star(graph, start, end)

print("Path found:", path)

**Output**



**Time Complexity: -**

Best case: - **O(V + E)**

Average case: - **O(V log V + E)**

Worst case: - **O(bd)**

**Program:- 5**

**Objective: -**

**The Min-Max algorithm is a decision-making algorithm primarily used in two-player, zero-sum games, such as chess, checkers, tic-tac-toe, and other strategic games. The algorithm simulates all possible moves in a game tree to determine the best move for a player, assuming that the opponent will always play optimally. It is fundamental in game theory and artificial intelligence (AI) for creating intelligent game-playing agents.**

**Code: -**

import math

def minimax (curDepth, nodeIndex,

maxTurn, scores,

targetDepth):

  if (curDepth == targetDepth):

      return scores[nodeIndex]

  if (maxTurn):

      return max(minimax(curDepth + 1, nodeIndex \* 2,

      False, scores, targetDepth),

      minimax(curDepth + 1, nodeIndex \* 2 + 1,

      False, scores, targetDepth))

  else:

      return min(minimax(curDepth + 1, nodeIndex \* 2,

      True, scores, targetDepth),

      minimax(curDepth + 1, nodeIndex \* 2 + 1,

      True, scores, targetDepth))

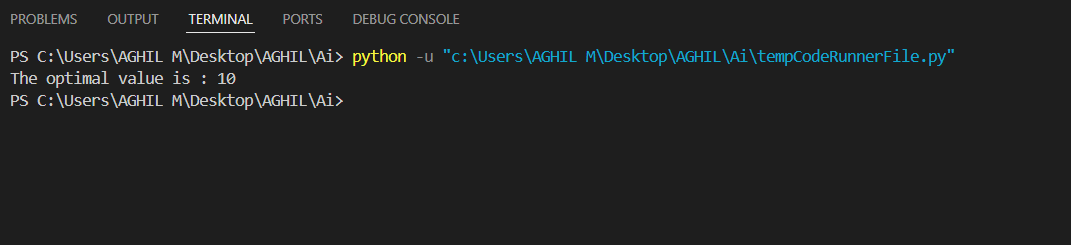
scores = [7 , 3 , 9 , 6 , 10, 5, 24, 23]

treeDepth = math.log(len(scores), 2)

print("The optimal value is : ", end = "")

print(minimax(0, 0, True, scores, treeDepth))

Output: -



**Time Complexity :-**

Best case: - **O(bd)**

Average case: - **O(bd)**

Worst case: - **O(bd)**

**Program: -6**

**Objective: -**

**Implement Alpha-beta pruning in python.**

**Code:**

import math

def alpha\_beta\_pruning(depth, node\_index, maximizing\_player, values, alpha, beta):

    if depth == 3:

        return values[node\_index]

    if maximizing\_player:

        max\_eval = -math.inf

        for i in range(2):

            eval = alpha\_beta\_pruning(depth + 1, node\_index \* 2 + i, False, values, alpha, beta)

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

            alpha = max(alpha, eval)

            if beta <= alpha:

                break

        return max\_eval

    else:

        min\_eval = math.inf

        for i in range(2):

            eval = alpha\_beta\_pruning(depth + 1, node\_index \* 2 + i, True, values, alpha, beta)

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

            beta = min(beta, eval)

            if beta <= alpha:

                break

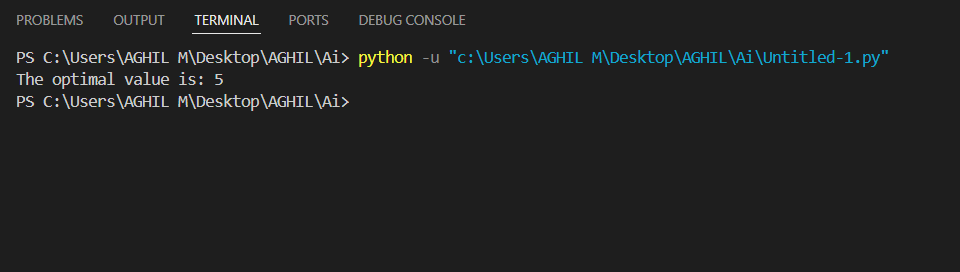
        return min\_eval

values = [3, 5, 6, 9, 1, 2, 0, -1]

optimal\_value = alpha\_beta\_pruning(0, 0, True, values, -math.inf, math.inf)

print("The optimal value is:",optimal\_value)

**Output: -**

****