

**CENTRAL UNIVERSITY OF RAJASTHAN**

**Department of Computer Science**

**NAME : Meka Bhoomika**

**ENROLLMENT NO. :2021IMSCS013**

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

**COURSE NAME : Artificial Intelligence Lab**

**COURSE CODE :CSC-407**

**SUBMITTED TO :Dr.Gaurav Meena**

|  |  |
| --- | --- |
| S.No. | Table of Contents |
| 1 | Breadth First Search |
| 2 | Depth First Search |
| 3 | Best First Search |
| 4 | A\* Search |
| 5 | AO\* Search |
| 6 | Min -Max Pruning |
| 7 | Alpha-Beta Pruning |
|  |  |
|  |  |

**PROGRAM -1**

Implement breadth first search in python.

**Objective :** [Breadth-first search (BFS) is a graph traversal algorithm that finds shortest paths from a given source vertex to all other vertices](https://www.bing.com/ck/a?!&&p=1f13df1e8f7be40053ed887d4726f962d47d1c516f225a387a58474f7bb57452JmltdHM9MTczMzE4NDAwMA&ptn=3&ver=2&hsh=4&fclid=1df5c38b-a89d-65a6-2cf3-d695a99b6484&psq=objective+for+breadth+first+search&u=a1aHR0cHM6Ly93d3cua2hhbmFjYWRlbXkub3JnL2NvbXB1dGluZy9jb21wdXRlci1zY2llbmNlL2FsZ29yaXRobXMvYnJlYWR0aC1maXJzdC1zZWFyY2gvYS9icmVhZHRoLWZpcnN0LXNlYXJjaC1hbmQtaXRzLXVzZXM&ntb=1).

**Code:**

graph = {

  '12' : ['13','7'],

  '13' : ['2', '4'],

  '7' : ['18'],

  '2' : [],

  '4' : ['18'],

  '18' : []

}

visited = []

queue = []

def bfs(visited, graph, node):

  visited.append(node)

  queue.append(node)

  while queue:

    m = queue.pop(0)

    print (m, end = " ")

    for neighbour in graph[m]:

      if neighbour not in visited:

        visited.append(neighbour)

        queue.append(neighbour)

print("Following is the Breadth-First Search")

bfs(visited, graph, '12')

Output:



**TIME COMPLEXITY ANALYSIS :**

**BEST CASE :**  The best case occurs when the goal node is found early, e.g., the root or its immediate children. In this scenario, the algorithm visits only a small subset of the graph before stopping.

Therefore, O(V) time complexity.

**Worst case :** In the worst case, the algorithm explores all vertices (V) and edges (E) because the goal is not present or is located at the last possible node in the traversal.

Therefore, O(V+E) time complexity.

**Average case :** On average, DFS explores most of the graph due to the structure of the traversal, as it does not prioritize closer nodes. The complexity depends on the graph's density.

Therefore, O(V+E) time complexity.

**PROGRAM -2**

Implement depth first search in python.

**Objective : :** Depth First Search (DFS) is an algorithm used for traversing or searching tree and graph data structures.

**Code:**

graph = {

  '12' : ['13','7'],

  '13' : ['2', '4'],

  '7' : ['18'],

  '2' : [],

  '4' : ['18'],

  '18' : []

}

visited = set()

def dfs(visited, graph, node):

    if node not in visited:

        print (node)

        visited.add(node)

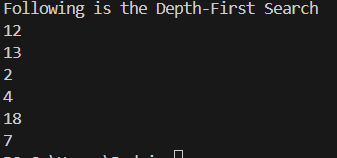
        for neighbour in graph[node]:

            dfs(visited, graph, neighbour)

print("Following is the Depth-First Search")

dfs(visited, graph, '12')

Output:



***TIME COMPLEXITY ANALYSIS :***

***BEST CASE :*** In Occurs when the graph is sparse (few edges) and the traversal stops early, visiting only a subset of nodes.

Therefore, O(V) time complexity.

***Worst case :***  Occurs when the graph is dense or all edges and vertices are traversed, as DFS visits all vertices and explores all edges.

Therefore, O(V+E) time complexity.

***Average case :*** On average, DFS visits all vertices and edges reachable from the starting node.

Therefore, O(V+E) time complexity.

**PROGRAM -3**

Implement Best First Search algorithm in python.

**Code:**

from queue import PriorityQueue

v = 14

graph = [[] for i in range(v)]

def best\_first\_search(actual\_src, target, n):

    visited = [False] \* n

    pq = PriorityQueue()

    pq.put((0, actual\_src))

    visited[actual\_src] = True

    while not pq.empty():

        u = pq.get()[1]

        print(u, end=" ")

        if u == target:

            break

        for v, c in graph[u]:

            if not visited[v]:

                visited[v] = True

                pq.put((c, v))

def addedge(x, y, cost):

    graph[x].append((y, cost))

    graph[y].append((x, cost))

addedge(0, 1, 3)

addedge(0, 2, 9)

addedge(0, 3, 5)

addedge(1, 4, 9)

addedge(1, 2, 8)

addedge(2, 6, 12)

addedge(2, 7, 14)

addedge(3, 8, 7)

addedge(8, 9, 5)

addedge(8, 10, 6)

addedge(9, 11, 1)

addedge(9, 12, 10)

addedge(9, 13, 2)

source = 0

taret = 9

print("Best-First Search Path:")

best\_first\_search(source, target, v)

**Output :**

****

**Time Complexity:**

* **Best-case:** O(V + E).
* **Average-case:** O(V log V + E).
* **Worst-case:** O(b^d).

**PROGRAM -4**

Implement A\* Search algorithm in python.

**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

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

      if neighbor not in [node for \_, node, \_ in open\_set] or tentative\_g\_score < f\_score:

        new\_f\_score = tentative\_g\_score + heuristic(neighbor, goal)

        heapq.heappush(open\_set, (new\_f\_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 // 2 - goal // 2) + abs(node % 2 - goal % 2)

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 -5**

Implement AO\* Search algorithm in python.

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

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

      if neighbor not in [node for \_, node, \_ in open\_set] or tentative\_g\_score < f\_score:

        new\_f\_score = tentative\_g\_score + heuristic(neighbor, goal)

        heapq.heappush(open\_set, (new\_f\_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 // 2 - goal // 2) + abs(node % 2 - goal % 2)

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+E)

Average-case: O(VlogV+E)

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

**PROGRAM -6**

Implement Minimax algorithm in python.

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))

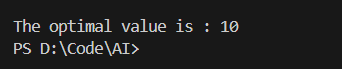
scores = [7 , 3 , 9 , 6 , 10, 5, 24, 23]

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 -7**

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:



The time complexity of the Alpha-Beta Pruning algorithm is O(b^(d/2)), where:

• b is the branching factor (the average number of child nodes per node).

• d is the depth of the game tree.