**Name:- CHANDAN GORAI**

**SEC-A**

**REG No:-2301292197**

**Breadth First Search (BFS):-**

from collections import deque

# 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']

}

# Initialize the starting node and data structures

start = 'A'

queue = deque([start]) # Create a queue and enqueue the start node

visited = set([start]) # Create a set to track visited nodes

# Perform BFS

while queue:

node = queue.popleft() # Dequeue a node from the front of the queue

print(node, end=" ") # Process the node (here we print it)

# Explore the neighbors of the current node

for neighbor in graph[node]:

if neighbor not in visited:

visited.add(neighbor) # Mark the neighbor as visited

queue.append(neighbor) # Enqueue the neighbor

**Output for this example:**

A B C D E F

**DFS Algorithm and Code**

**Python Code for DFS:-**

**# Graph represented as an adjacency list**

**graph = {**

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

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

**'C': ['G'],**

**'D': [],**

**'E': [],**

**'F': [],**

**'G': []**

**}**

**# Initialize**

**start\_node = 'A'**

**visited = set() # To track visited nodes**

**stack = [start\_node] # Stack to store nodes to visit**

**# Depth First Search (without recursion)**

**print("DFS Traversal Order:")**

**while stack:**

**current\_node = stack.pop() # Get the last added node**

**if current\_node not in visited:**

**print(current\_node, end=" ") # Process the current node**

**visited.add(current\_node) # Mark as visited**

**# Add neighbors in reverse order to maintain the correct order**

**for neighbor in reversed(graph[current\_node]):**

**if neighbor not in visited:**

**stack.append(neighbor)**

**Python Code [Best first search]:-**

**import heapq**

**# Graph represented as an adjacency list**

**graph = {**

**0: [1, 2],**

**1: [0, 3, 4],**

**2: [0, 5],**

**3: [1],**

**4: [1],**

**5: [2]**

**}**

**# Heuristic values for each node**

**heuristic = {**

**0: 10,**

**1: 8,**

**2: 5,**

**3: 7,**

**4: 3,**

**5: 0 # Assuming node 5 is the goal**

**}**

**# Initialization**

**start\_node = 0**

**goal\_node = 5**

**priority\_queue = []**

**heapq.heappush(priority\_queue, (heuristic[start\_node], start\_node)) # (heuristic, node)**

**visited = set()**

**path = []**

**# Best-First Search**

**while priority\_queue:**

**# Get the node with the smallest heuristic value**

**\_, current = heapq.heappop(priority\_queue)**

**if current in visited:**

**continue**

**# Mark the current node as visited and add it to the path**

**visited.add(current)**

**path.append(current)**

**# If the goal is reached, stop**

**if current == goal\_node:**

**break**

**# Add neighbors to the priority queue**

**for neighbor in graph[current]:**

**if neighbor not in visited:**

**heapq.heappush(priority\_queue, (heuristic[neighbor], neighbor))**

**# Output the result**

**print("Best-First Search Traversal:", path)**

**Output:**

**Best-First Search Traversal: [0, 2, 5]**

***A*\* *Search Algorithm***

**Inputs:**

1. **Graph: A tree with nodes and edges where each edge has a cost.**
2. **Start node: The initial node where the search begins.**
3. **Goal node: The destination node that we want to reach.**
4. **Heuristic: A heuristic function or pre-defined values estimating the cost from each node to the goal node.**

**Outputs:**

* **The optimal path from the start node to the goal node, or a message saying no path exists.**

**Step-by-Step Algorithm:**

1. **Initialization:**
   * **Define the open list: A priority queue (min-heap) which will hold nodes that need to be evaluated.**
   * **Define the closed list: A set to keep track of nodes that have already been evaluated.**
   * **Define Actual cost(g-cost) for each node, which represents the cost from the start node to the current node.**
   * **Define heuristic values(h value) for each node, which estimate the cost from that node to the goal node.**
2. **Start:**
   * **Add the start node to the open list with its F Value.**
3. **Main Loop:**
   * **While the open list is not empty:**
     1. **Select the node from the open list that has the lowest F Value. This node is the most promising candidate for exploration.**
     2. **If this node is the goal node, the search is complete. Reconstruct the path from the goal node back to the start node.**
     3. **Otherwise, move the current node from the open list to the closed list, marking it as explored.**
     4. **Explore each neighbor of the current node:**
        + **For each neighbor, calculate the tentative g-cost: the cost to reach that neighbor through the current node.**
        + **If the neighbor has not been evaluated (i.e., it’s not in the closed list), or if a better path to that neighbor is found (i.e., the tentative g-value is lower than the current g-value), then:**
          - **Update the g-value for that neighbor.**
          - **Calculate the new F value.**
          - **If the neighbor is not already in the open list, add it.**
4. **Termination:**
   * **If the goal node is found, return the reconstructed path.**
   * **If the open list becomes empty and no path to the goal node is found, return that no path exists.**

**Python Code for A\* Search:--**

**import heapq**

**# Node and heuristic setup**

**start = "A" # Starting node**

**goal = "G" # Goal node**

**nodes = {**

**"A": {"B": 1, "C": 4}, # A has children B and C, costs 1 and 4 respectively**

**"B": {"D": 2, "E": 5}, # B has children D and E, costs 2 and 5 respectively**

**"C": {"E": 1}, # C has child E, cost 1**

**"D": {"G": 3}, # D has child G, cost 3**

**"E": {"G": 1}, # E has child G, cost 1**

**"G": {} # G has no children, it's the goal**

**}**

**# Heuristic values (estimated cost to reach the goal)**

**heuristic = {**

**"A": 6, # Estimated cost to reach G from A**

**"B": 5,**

**"C": 2,**

**"D": 1,**

**"E": 1,**

**"G": 0 # The goal node has a heuristic of 0**

**}**

**# Open list (priority queue) and closed list**

**open\_list = []**

**closed\_list = set()**

**# Starting node setup**

**heapq.heappush(open\_list, (heuristic[start], 0, start)) # (f\_score, g\_score, node)**

**g\_score = {start: 0} # Starting node has a g\_score of 0**

**came\_from = {} # To reconstruct the path**

**while open\_list:**

**# Pop the node with the lowest f\_score**

**current\_f, current\_g, current\_node = heapq.heappop(open\_list)**

**# If the current node is the goal, reconstruct the path**

**if current\_node == goal:**

**path = []**

**while current\_node in came\_from:**

**path.append(current\_node)**

**current\_node = came\_from[current\_node]**

**path.append(start)**

**path.reverse()**

**print("Path found:", path)**

**break**

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

**# Explore neighbors**

**for neighbor, cost in nodes[current\_node].items():**

**if neighbor in closed\_list:**

**continue**

**tentative\_g\_score = current\_g + cost**

**# If the neighbor is not in the open list or we found a better path to it**

**if neighbor not in g\_score or tentative\_g\_score < g\_score[neighbor]:**

**g\_score[neighbor] = tentative\_g\_score**

**f\_score = tentative\_g\_score + heuristic[neighbor]**

**heapq.heappush(open\_list, (f\_score, tentative\_g\_score, neighbor))**

**came\_from[neighbor] = current\_node**

**else:**

**print("No path found.")**

**OUTPUT:-**

**Path found: ['A', 'B', 'D', 'G']**