DFS

Step 1: Initialize the Open and Closed Lists

- **Open List**: This is a stack data structure used to keep track of nodes that need to be explored. We start by adding the `start_node` to this list.
- **Closed List**: This is a set used to keep track of nodes that have already been visited to avoid re-processing them.
- Path Dictionary: Maps each node to its predecessor to reconstruct the path later

Step 2: Loop Until Open List is Empty

- The main loop runs as long as there are nodes in the open list.

Step 3: Pop a Node from the Open List

- The last node (most recently added) is removed from the open list. This ensures that we are exploring as deep as possible along each branch before backtracking.

Step 4: Check if the Current Node is the Goal Node

- If the current node is the goal node, we can stop the search and return a message or the path found.

Step 5: Process the Current Node if Not Visited

- If the current node is not in the closed list (i.e., not visited):
  - Add it to the closed list.
  - Retrieve its adjacent nodes (children).

Step 6: Add Children to the Open List

- For each child of the current node:
  - If the child has not been visited (not in the closed list), add it to the open list.
  - This ensures that unvisited children will be explored in subsequent iterations.

Step 7: Repeat Until Open List is Empty or Goal is Found

- Continue the loop until there are no more nodes to explore or the goal node is found.

Step 8: **Reconstruct Path Function**:

- If the goal node is found, backtrack from the `goal_node` to the `start_node` using the path dictionary.

- Reverse the path to get it in the correct order from `start_node` to `goal_node`.

Step 9: Return Result:

- Return the path if found, otherwise indicate that no path exists.

```
def depth_first_search(graph, start_node, goal_node):
    # Open list: stack to keep track of nodes to explore
    open_list = [start_node]
    # Closed list: set to keep track of visited nodes
    closed_list = set()
    # Dictionary to store the path
    path = {start_node: None}

    while open_list:
        # Pop the last node from the stack
        current_node = open_list.pop()

        # If the goal node is found, reconstruct the path
        if current_node == goal_node:
            return reconstruct_path(path, start_node, goal_node)

        # If the current node has not been visited
        if current_node not in closed_list:
            # Add the current node to the closed list
            closed_list.add(current_node)

            # Get all adjacent nodes (children) of the current node
            children = graph.get(current_node, [])

            # Add each child to the open list (stack) if not visited
            for child in children:
                if child not in closed_list:
                    open_list.append(child)
                    # Store the path
                    path[child] = current_node
```

```
    return f"Goal node {goal_node} not found in the graph."

def reconstruct_path(path, start_node, goal_node):
    # Reconstruct the path from goal_node to start_node
    current_node = goal_node
    reversed_path = []
    while current_node is not None:
        reversed_path.append(current_node)
        current_node = path[current_node]
    # Return the reversed path (from start_node to goal_node)
    return list(reversed(reversed_path))

# Example graph represented as an adjacency list
graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F'],
    'D': [],
    'E': ['G'],
    'G': [],
}

# Perform DFS
start_node = 'A'
goal_node = 'F'
result = depth_first_search(graph, start_node, goal_node)
print("Path from start to goal: ",result)  # Output: ['A', 'C', 'F']
```

OUTPUT

```
Path from start to goal: ['A', 'C', 'F']
```

EXP NO: 3 UNINFORMED SEARCHES

3.1. Breadth First Search Algorithm

1. **START**
2. **Enqueue the Start State**:
   - Enqueue the start state into the queue.
   - Mark the start state as visited by adding it to the visited set.
   - Initialize the predecessor of the start state as `None`
3. **While the Queue is Not Empty**:
   - Dequeue a node(`current_node`) from the front of the queue.
   - If `current_node` is the goal state, break the loop
4. **For Each Neighbor of the `current_node`**:
   - If the neighbor has not been visited:
     - Enqueue the neighbor.
     - Mark the neighbor as visited.
     - Set the predecessor of the neighbor as the `current_node`.
5. If the goal state is reached, reconstruct the path from the start state to the goal state using the predecessor dictionary.
   - **Reconstruction of path**: Initialize an empty list to store the path.
     - If the goal node was visited:
     - Starting from the goal node, trace back to the start node using the predecessor dictionary.
     - Append each node to the path list.
     - Reverse the path list to get the correct order from start to goal.
6. Return the path from the start state to the goal state if found, otherwise return a message indicating no path exists.
7. **STOP**

```
from collections import deque


def bfs(graph, start, goal):
    # Initialize the queue, visited set, and predecessor dictionary
    queue = deque([start])
    visited = set([start])
    predecessor = {start: None}

    # While the queue is not empty
    while queue:
        current_node = queue.popleft()
```

```
        # If the goal state is found, break the loop
        if current_node == goal:
            break

        # For each neighbor of the current node
        for neighbor in graph[current_node]:
            if neighbor not in visited:
                queue.append(neighbor)
                visited.add(neighbor)
                predecessor[neighbor] = current_node

    # Path reconstruction
    path = []
    if goal in visited:
        current = goal
        while current is not None:
            path.append(current)
            current = predecessor[current]
        path.reverse()
        return path
    else:
        return "No path exists"

# Example usage
graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F'],
    'D': [],
    'E': ['G'],
```

```
    'G': []
}

start = 'A'
goal = 'F'
path = bfs(graph, start, goal)
print("Path from start to goal:", path)
```

OUTPUT

```
Path from start to goal: ['A', 'C', 'F']
```

1. START
2. **While Open List is Not Empty**:

   Pop Node with Minimum `f` Value:

   - Remove the node from the open list that has the smallest `f` value (`g + heuristic`). This node is considered the most promising to expand next.
   - Print the name of the node being expanded for tracking purposes.

   Check for Goal Node:

   - If the current node is the `goal` node:
     - Print a message indicating that the goal has been found.
     - Call the `reconstruct_path` function to retrieve the path from the start to the goal and return it.

   Mark Node as Explored:

   - Add the current node to the closed list to mark it as fully explored.

   Expand Neighbors:

   - For each neighbor of the current node:
     - **Skip if Neighbor is in Closed List**:
       - If the neighbor has already been explored (i.e., is in the closed list), skip it.
     - **Calculate Tentative `g` Value**:
       - Calculate the tentative cost (`tentative_g`) to reach the neighbor node as the sum of the current node's cost (`g`) and the cost to reach this neighbor.
     - **Update Neighbor's Cost and Parent**:
       - If the neighbor is not in the open list or the newly calculated cost (`tentative_g`) is lower than the neighbor's current cost (`g`):
         - Update the neighbor's cost (`g`) to the `tentative_g`.
         - Set the parent of the neighbor to the current node.
         - If the neighbor is not already in the open list, add it to the open list.
3. **If Open List is Empty**:If the open list becomes empty and the goal has not been reached: Print a message indicating that the goal node was not found.
4. **Reconstruct Path**:

   Trace Back from Goal Node:Start from the `goal` node and trace back to the start node using the `parent` references.

Build Path:Collect the names of the nodes in a list as you trace back.Reverse the list to get the path from the start node to the goal node.

Return Path:

Return the reconstructed path.

5. STOP

```
import heapq

class Node:
    def __init__(self, name, heuristic):
        self.name = name
        self.heuristic = heuristic
        self.neighbors = {}  # Dictionary of {neighbor_node: cost}
        self.g = float('inf')  # Cost from start to this node
        self.parent = None

    def add_neighbor(self, neighbor, cost):
        self.neighbors[neighbor] = cost

    def __lt__(self, other):
        # Compare nodes based on f = g + heuristic
        return (self.g + self.heuristic) < (other.g + other.heuristic)

def a_star_search(start, goal):
    open_list = []
    closed_list = set()

    start.g = 0
    heapq.heappush(open_list, start)

    while open_list:
        current_node = heapq.heappop(open_list)
        print(f"Expanding node: {current_node.name}")

        if current_node == goal:
            return reconstruct_path(goal)

        closed_list.add(current_node)

        for neighbor, cost in current_node.neighbors.items():
            if neighbor in closed_list:
                continue

            tentative_g = current_node.g + cost

            if tentative_g < neighbor.g:
                neighbor.g = tentative_g
                neighbor.parent = current_node
                if neighbor not in open_list:
                    heapq.heappush(open_list, neighbor)
```

```python
    print("Goal node not found.")
    return None

def reconstruct_path(goal):
    path = []
    current = goal
    while current:
        path.append(current.name)
        current = current.parent
    path.reverse()
    return path

# Example Usage
if __name__ == "__main__":
    # Create nodes with heuristic values
    a = Node('A', 5)
    b = Node('B', 3)
    c = Node('C', 1)
    d = Node('D', 2)
    e = Node('E', 0)

    # Add neighbors (node, cost)
    a.add_neighbor(b, 1)
    a.add_neighbor(c, 4)
    b.add_neighbor(d, 2)
    c.add_neighbor(e, 3)
    d.add_neighbor(e, 1)

    # Perform A* Search
    path = a_star_search(a, e)
    if path:
        print("Path found:", " -> ".join(path))
```

OUTPUT

```
Expanding node: A
Expanding node: B
Expanding node: C
Expanding node: D
Expanding node: E
Path found: A -> B -> D -> E
```

BEST FIRST SEARCH

1. START
2. Insert the `start` node into the open list with its heuristic value as the priority.
3. **While Open List is Not Empty**:
   a. **Expand Node**:
      i. Remove the node with the lowest heuristic value from the open list. This is the node that is currently considered the most promising to reach the goal.
      ii. Print or record the node being expanded for debugging or information purposes.
   b. **Check for Goal**:
      i. If the current node is the `goal` node, print that the goal has been found and terminate the search.
   c. **Mark Node as Explored**:
      i. Add the current node to the closed list to mark it as explored.
   d. **Process Neighbors**:
      i. For each neighbor of the current node:
         1. **If Neighbor is in Closed List**: Skip this neighbor since it has already been explored.
         2. **If Neighbor is Not in Open List**:
            a. Add the neighbor to the open list with its heuristic value as the priority.
         3. **If Neighbor is Already in Open List**: This step is usually implicit since the neighbor is added only if not already present.
4. If the open list is empty and the goal has not been found, print that the goal node is not reachable or not found.
5. STOP

```python
class Node:
    def __init__(self, name, heuristic):
        self.name = name
        self.heuristic = heuristic
        self.neighbors = {}

    def add_neighbor(self, neighbor, cost):
        self.neighbors[neighbor] = cost

def best_first_search(start, goal):
    open_list = [start]  # List of nodes to be explored
    closed_list = set()  # Set of nodes that have been explored

    while open_list:
        # Choose the node with the smallest heuristic value
        current_node = min(open_list, key=lambda node: node.heuristic)
        open_list.remove(current_node)

        print(f"Expanding node: {current_node.name}")
```

```
        # Check if the current node is the goal
        if current_node == goal:
            print(f"Goal node {goal.name} found!")
            return

        closed_list.add(current_node)

        # Add neighbors to the open list
        for neighbor in current_node.neighbors:
            if neighbor in closed_list or neighbor in open_list:
                continue
            open_list.append(neighbor)

    print("Goal node not found.")

# Example Usage
if __name__ == "__main__":
    # Creating nodes with heuristic values
    a = Node('A', 5)
    b = Node('B', 3)
    c = Node('C', 1)
    d = Node('D', 2)
    e = Node('E', 0)

    # Adding neighbors (node, cost)
    a.add_neighbor(b, 1)
    a.add_neighbor(c, 4)
    b.add_neighbor(d, 2)
    c.add_neighbor(e, 3)
    d.add_neighbor(e, 1)

    # Performing Best First Search
    best_first_search(a, e)
```

OUTPUT

```
Expanding node: A
Expanding node: C
Expanding node: E
Goal node E found!
```

EXP7. Game playing (adversarial search)

MIN MAX Game Playing with Alpha-Beta Pruning

AIM

Implement MINMAX Game Playing with Alpha-Beta Pruning

<u>ALGORITHM</u>

1. **Function Definition**:

   Define the function `minimax(node, depth, isMaximizingPlayer, alpha, beta)`.

   ```
   function minimax(node, depth, isMaximizingPlayer, alpha,
   beta):
   ```

2. **Base Case**:

   Check if the current `node` is a leaf node (i.e., no children):

   - If it is a leaf node, return the value of the node.

```
if node is a leaf node :
    return value of the node
```

3. **Maximizing Player Logic**:

```
if isMaximizingPlayer :
    bestVal = -INFINITY
    for each child node :
        value = minimax(node, depth+1, false, alpha, beta)
        bestVal = max( bestVal, value)
        alpha = max( alpha, bestVal)
        if beta <= alpha:
            break
    return bestVal
```

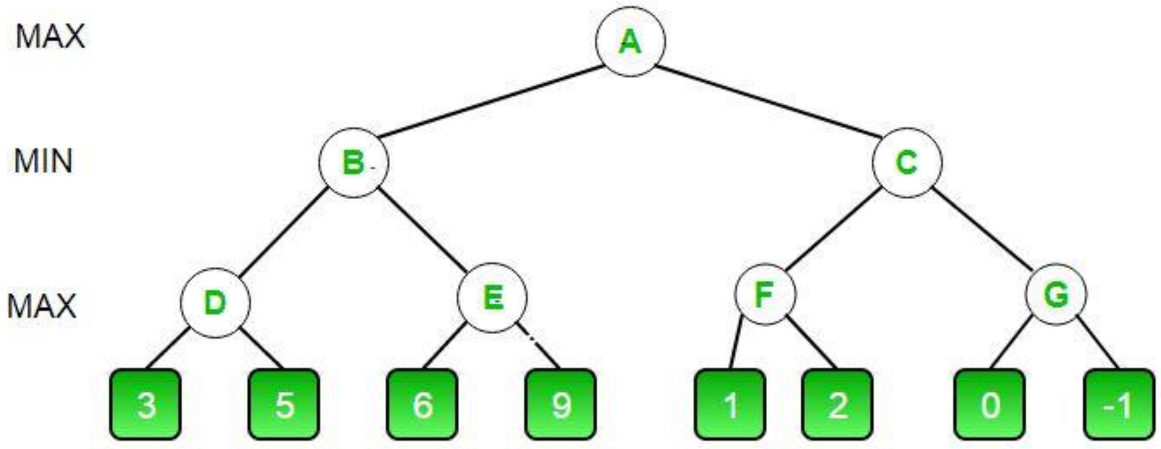
4. `else :` **Minimizing Player Logic**:

```
    bestVal = +INFINITY
    for each child node :
        value = minimax(node, depth+1, true, alpha, beta)
        bestVal = min( bestVal, value)
        beta = min( beta, bestVal)
        if beta <= alpha:
            break
    return bestVal
```

5. **Function Invocation**:

- To start the algorithm, call the `minimax` function with the following parameters:
  - The root node of the game tree.

- `depth` set to `0`.
- `isMaximizingPlayer` set to `True` (indicating it's the maximizing player's turn).
- `alpha` set to `-INFINITY`.
- `beta` set to `+INFINITY`.



PROGRAM

```
# Python3 program to demonstrate
# working of Alpha-Beta Pruning

# Initial values of Alpha and Beta
MAX, MIN = 1000, -1000

# Returns optimal value for current player
#(Initially called for root and maximizer)
def minimax(depth, nodeIndex, maximizingPlayer,
            values, alpha, beta):

    # Terminating condition. i.e
    # leaf node is reached
    if depth == 3:
        return values[nodeIndex]
```

```
if maximizingPlayer:

    best = MIN

    # Recur for left and right children
    for i in range(0, 2):

        val = minimax(depth + 1, nodeIndex * 2 + i,
                      False, values, alpha, beta)
        best = max(best, val)
        alpha = max(alpha, best)

        # Alpha Beta Pruning
        if beta <= alpha:
       print("best value of max player-->",best)

            break

    return best

else:
    best = MAX

    # Recur for left and
    # right children
    for i in range(0, 2):

        val = minimax(depth + 1, nodeIndex * 2 + i,
                        True, values, alpha, beta)
        best = min(best, val)
        beta = min(beta, best)
```

```
            # Alpha Beta Pruning
            if beta <= alpha:
                break
           print("best value of min player-->",best)

        return best

# Driver Code
if __name__ == "__main__":

    values = [3, 5, 6, 9, 1, 2, 0, -1]
    print("The optimal value is :", minimax(0, 0, True, values, MIN, MAX))
```

RESULT

The above program has been successfully executed and output obtained is verified.

OUTPUT

The optimal value is : 5