Practical No 1

Aim: Program to implement Uninformed Search Technique: Breadth First Search

```
from collections import deque

def bfs(graph, start, goal):

    visited = set()  # Set to keep track of visited nodes

    queue = deque()  # Queue for BFS

    # Add the start node to the queue and mark it as visited

    queue.append(start)

    visited.add(start)

    while queue:

        current_node = queue.popleft()

        print(current_node, end=" ")

        # Check if we have reached the goal node

        if current_node == goal:

            print("\nGoal reached!")

            return True

        # Add all adjacent nodes of the current node to the queue

        for neighbor in graph[current_node]:

            if neighbor not in visited:

                queue.append(neighbor)

                visited.add(neighbor)

    print("\nGoal not found!")

    return False

# Define the graph as a dictionary of lists

graph = {

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

```
    'B': ['D', 'E'],
    'C': ['F'],
    'D': [],
    'E': ['F'],
    'F': [],
}
# Call the bfs function
bfs(graph, 'A', 'F')
```

Output:

```
>>>
    =============== RESTART: C:\Users\admin\Downloads\Practical 1.py ===============
    A B C D E F
    Goal reached!
>>>
```

Practical No 2

Aim: - Program to implement Uninformed Search Technique: Depth First Search

```
class Node:
    def __init__(self, value):
        self.value = value
        self.children = []
def dfs(node, target):
    if node.value == target:
        return True
    for child in node.children:
        if dfs(child, target):
            return True
    return False
# Create nodes
A = Node('A')
B = Node('B')
C = Node('C')
D = Node('D')
E = Node('E')
F = Node('F')
G = Node('G')
# Connect nodes
A.children = [B, C, D]
B.children = [E, F]
D.children = [G]
# Perform DFS
```

```
print(dfs(A, 'F'))
print(dfs(A, 'G'))
print(dfs(A, 'H'))
```

Output:

```
>>>
    =============== RESTART: C:\Users\admin\Downloads\practical 2.py ========
    True
    True
    False
>>>
```

Practical No 3

Aim: - Program to implement Informed Search Technique: A* Algorithm

```
import heapq

def astar(maze, start, goal):

    # Heuristic function (Manhattan distance)

    def h(node):

        return abs(node[0] - goal[0]) + abs(node[1] - goal[1])

    # Open list (priority queue), closed list, and the 'g' dictionary (cost and parent)

    o, c, g = [(0, start)], set(), {start: (0, None)}

    while o:

        # Get the node with the lowest f value (g + h)

        f, n = heapq.heappop(o)

        # If we reach the goal, reconstruct the path

        if n == goal:

            path = []

            while n is not None:

                path.append(n)

                n = g[n][1]

            return path[::-1]

        if n in c:

            continue

        # Mark the node as visited

        c.add(n)

        # Check the 4 neighboring nodes (up, down, left, right)

        for i, j in [(1, 0), (0, 1), (-1, 0), (0, -1)]:

            nb = n[0] + i, n[1] + j

            # Check if the neighbor is within bounds and walkable (0)
```

```
            if 0 <= nb[0] < len(maze) and 0 <= nb[1] < len(maze[0]) and maze[nb[0]][nb[1]] ==
0:
                gs = g[n][0] + 1  # g(n) is the cost to reach the neighbor
                # If the neighbor is not in the 'g' dictionary or we found a better path to it
                if nb not in g or gs < g[nb][0]:
                    g[nb] = (gs, n)
                    heapq.heappush(o, (gs + h(nb), nb))  # Add to open list with priority (f = g + h)
    return []  # Return an empty path if no path is found
# Example usage:
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, end = (0, 0), (7, 6)
path = astar(maze, start, end)
print(path)
```

Output:

```
>>>
=============== RESTART: C:\Users\admin\Downloads\practical 3.py ===============
[(0, 0), (0, 1), (0, 2), (0, 3), (1, 3), (2, 3), (3, 3), (4, 3), (5, 3), (5, 4), (5, 5), (5, 6), (6, 6), (7, 6)]
>>>
```

Practical No 4

Aim: - Program to implement Game Playing Algorithms: Minimax and Alpha Beta Pruning

```
MAX, MIN = 1000, -1000

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(2):  # i = 0, 1 for left and right children

            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:

                break

        return best

    else:

        best = MAX

        # Recur for left and right children

        for i in range(2):  # i = 0, 1 for left and right children

            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
        return best
# Example values for the leaf nodes of the game tree
values = [3, 5, 6, 9, 1, 2, 0, -1]
print("The optimal value is:", minimax(0, 0, True, values, MIN, MAX))
```

Output:

```
>>>
=============== RESTART: C:\Users\admin\Downloads\Practical 4.py ==========
The optimal value is: 5
>>>
```

Practical No 5

Aim: - To Implement Forward Chaining Algorithm.

```
# Rules for forward chaining
rules = {
    'rule1': [['A'], ['B', 'C']],
    'rule2': [['B', 'C'], ['D']],
    'rule3': [['B'], ['E']],
    'rule4': [['E'], ['F']]
}
# Initialize the knowledge base with known facts
knowledge_base = {
    'A': True
}
# Function to perform forward chaining
def forward_chaining(kb, rules):
    changed = True  # Flag to indicate if new facts are added to the KB
    while changed:
        changed = False
        for rule, (antecedent, consequent) in rules.items():
            # Check if all the antecedents are true in the knowledge base
            if all(kb.get(fact) for fact in antecedent):
                # If the consequent is not already in the knowledge base, add it
                for fact in consequent:
                    if fact not in kb:
                        kb[fact] = True
                        changed = True  # Set changed to True if a new fact is added
```

```
# Run forward chaining

forward_chaining(knowledge_base, rules)

# Print the updated knowledge base

print("Updated Knowledge Base:")

for fact, value in knowledge_base.items():

    print(f"{fact}: {value}")
```

Output:

```
>>
   =============== RESTART: C:\Users\admin\Downloads\practical 5.py ===============
   Updated Knowledge Base:
   A: True
   B: True
   C: True
   D: True
   E: True
   F: True
>>
```

Practical No 6

Aim: - To Implement Backward Chaining Algorithm.

```
# Define the facts and rules for backward chaining
facts = {
    'a': True,
    'b': False,
    'c': True,
    'd': False
}
rules = {
    'r1': {'a', 'b'},  # Rule r1 requires 'a' and 'b'
    'r2': {'c', 'd'},  # Rule r2 requires 'c' and 'd'
    'r3': {'a'}        # Rule r3 requires 'a'
}
# Function to perform backward chaining
def backward_chaining(target):
    # If the target is a fact, return its value
    if target in facts:
        return facts[target]
    # If the target is a rule, try to apply the rule by checking its prerequisites
    if target in rules:
        for fact in rules[target]:  # For each fact in the rule's antecedent
            if not backward_chaining(fact):  # If any fact fails, return False
                return False
        return True  # If all facts in the rule are true, the target is true
    return False  # Return False if the target isn't found in facts or rules
```

Test the backward chaining algorithm

query = 'r2'

result = backward_chaining(query)

print(f"The result for query '{query}' is: {result}")

<u>Output:</u>

```
>>>
=============== RESTART: C:\Users\admin\Downloads\practical 6.py ============
The result for query 'r2' is: False
>>>
```