Department of Computer Science and Engineering (Data Science)

Subject: Artificial Intelligence (DJS22DSC502)

AY: 2024-25

Experiment 2

(Uninformed Search)

DHRUV SHAH 60009220132 D11

Aim: Implement Depth First Iterative Deepening to find the path for a given planning problem.

Theory:

Solving a problem by search is solving a problem by trial and error. Several real-life problems can be modelled as a state-space search problem.

1. Choose your problem and determine what constitutes a STATE (a symbolic representation of the state-of-existence).

2. Identify the START STATE and the GOAL STATE(S).

3. Identify the MOVES (single-step operations/actions/rules) that cause a STATE to change.

4. Write a function that takes a STATE and applies all possible MOVES to that STATE to produce a set of NEIGHBOURING STATES (exactly one move away from the input state). Such a function (state-transition function) is called MoveGen. MoveGen embodies all the single-step operations/actions/rules/moves possible in a given STATE. The output of MoveGen is a set of NEIGHBOURING STATES. MoveGen: STATE -> SET OF NEIGHBOURING STATES. From a graph theoretic perspective the state space is a graph, implicitly defined by a MoveGen function. Each state is a node in the graph, and each edge represents one move that leads to a neighbouring state. Generating the neighbours of a state and adding them as candidates for inspection is called “expanding a state”. In state space search, a solution is found by exploring the state space with the help of a MoveGen function, i.e., expand the start state and expand every candidate until the goal state is found.

State spaces are used to represent two kinds of problems: configuration and planning problems.

1. In configuration problems the task is to find a goal state that satisfies some properties.

2. In planning problems the task is to find a path to a goal state. The sequence of moves in the path constitutes a plan.

Algorithm DFID

```
DFID-2(S)
1  count ← −1
2  path ← empty list
3  depthBound ← 0
4  repeat
5       previousCount ← count
6       (count, path) ← DB-DFS-2(S, depthBound)
7       depthBound ← depthBound + 1
8  until (path is not empty) or (previousCount = count)
9  return path

DB-DFS-2(S, depthBound)
    ▷ Opens new nodes, i.e., nodes neither in OPEN nor in CLOSED,
    ▷ and reopens nodes present in CLOSED and not present in OPEN.
10 count ← 0
11 OPEN ← (S, null, 0) : []
12 CLOSED ← empty list
13 while OPEN is not empty
14      nodePair ← head OPEN
15      (N, _, depth) ← nodePair
16      if GoalTest(N) = TRUE
17           return (count, ReconstructPath(nodePair, CLOSED))
18      else CLOSED ← nodePair : CLOSED
19           if depth < depthBound
20                neighbours ← MoveGen(N)
21                newNodes ← RemoveSeen(neighbours, OPEN, [])
22                newPairs ← MakePairs(newNodes, N, depth + 1)
23                OPEN ← newPairs ++ tail OPEN
24                count ← count + length newPairs
25           else OPEN ← tail OPEN
26 return (count, empty list)
```

Auxiliary Functions for DFID

```
MakePairs(nodeList, parent, depth)
1  if nodeList is empty
2       return empty list
3  else nodePair ← (head nodeList, parent, depth)
4       return nodePair : MakePairs(tail nodeList, parent, depth)
```

```
ReconstructPath(nodePair, CLOSED)
1  SkipTo(parent, nodePairs, depth)
2       if (parent, _, depth) = head nodePairs
3            return nodePairs
4       else return SkipTo(parent, tail nodePairs, depth)

5  (node, parent, depth) ← nodePair
6  path ← node : []
7  while parent is not null
8       path ← parent : path
9       CLOSED ← SkipTo(parent, CLOSED, depth – 1)
10      (_, parent, depth) ← head CLOSED
11 return path
```

Lab Assignment to do:

Select any one problem from the following and implement DFID to find the path from start state to goal state. Analyse the Time and Space complexity. Comment on Optimality and completeness of the solution.

Problem 1: 8-Puzzle

Problem 2: Water Jug

Problem 3: Graph

```
# 8-puzzle
def generate_moves(puzzle_state):
    possible_moves = []
    empty_tile_index = puzzle_state.index(0)

    def swap_positions(puzzle_state, i, j):
        state_list = list(puzzle_state)
        state_list[i], state_list[j] = state_list[j], state_list[i]
        return tuple(state_list)

    if empty_tile_index > 2:
        possible_moves.append(swap_positions(puzzle_state, empty_tile_index, empty_tile_index - 3))
    if empty_tile_index < 6:
        possible_moves.append(swap_positions(puzzle_state, empty_tile_index, empty_tile_index + 3))
    if empty_tile_index % 3 > 0:
        possible_moves.append(swap_positions(puzzle_state, empty_tile_index, empty_tile_index - 1))
    if empty_tile_index % 3 < 2:
        possible_moves.append(swap_positions(puzzle_state, empty_tile_index, empty_tile_index + 1))

    return possible_moves


# Goal test function
def goal_test(current_state, goal_state):
    return current_state == goal_state

# Depth-limited DFS function
def depth_limited_search(current_state, goal_state, depth_limit, path, visited):
    if current_state == goal_state:
        return path

    if depth_limit == 0:
        return None

    visited.add(current_state)
    for move in generate_moves(current_state):
        if move not in visited:
            result = depth_limited_search(move, goal_state, depth_limit - 1, path + [move], visited)
            if result:
                return result
    visited.remove(current_state)
    return None

# DFID function
def dfid(initial_state, goal_state):
    depth = 0
    while True:
        visited = set()
        result = depth_limited_search(initial_state, goal_state, depth, [initial_state], visited)
        if result is not None:
            return result
        depth += 1


user_input = input("Enter the initial state of the 8-puzzle (e.g., 1 2 3 4 0 5 6 7 8): ")
initial_puzzle_state = tuple(map(int, user_input.split()))
goal_puzzle_state = (1, 2, 3, 4, 5, 6, 7, 8, 0)

# DFID search for solution path
if goal_test(initial_puzzle_state, goal_puzzle_state):
    print("The initial state is already the goal state!")
else:
    solution_path = dfid(initial_puzzle_state, goal_puzzle_state)
    if solution_path:
        print("Path to goal state found!")
        for step, state in enumerate(solution_path):
            print(f"Step {step}: {state}")
    else:
        print("No solution found within reasonable depth limits.")
```

```
Enter the initial state of the 8-puzzle (e.g., 1 2 3 4 0 5 6 7 8): 1 2 3 4 0 5 6 7 8
Path to goal state found!
Step 0: (1, 2, 3, 4, 0, 5, 6, 7, 8)
Step 1: (1, 2, 3, 4, 5, 0, 6, 7, 8)
Step 2: (1, 2, 3, 4, 5, 8, 6, 7, 0)
Step 3: (1, 2, 3, 4, 5, 8, 6, 0, 7)
Step 4: (1, 2, 3, 4, 5, 8, 0, 6, 7)
Step 5: (1, 2, 3, 0, 5, 8, 4, 6, 7)
Step 6: (1, 2, 3, 5, 0, 8, 4, 6, 7)
Step 7: (1, 2, 3, 5, 6, 8, 4, 0, 7)
Step 8: (1, 2, 3, 5, 6, 8, 4, 7, 0)
Step 9: (1, 2, 3, 5, 6, 0, 4, 7, 8)
Step 10: (1, 2, 3, 5, 0, 6, 4, 7, 8)
```

```
Step 11: (1, 2, 3, 0, 5, 6, 4, 7, 8)
Step 12: (1, 2, 3, 4, 5, 6, 0, 7, 8)
Step 13: (1, 2, 3, 4, 5, 6, 7, 0, 8)
Step 14: (1, 2, 3, 4, 5, 6, 7, 8, 0)
```

Start coding or generate with AI.