Department of Computer Science and Engineering (Data Science)

Subject: Artificial Intelligence (DJ19DSC502)

AY: 2023-24

Experiment 2

(Uninformed Search)

Name: Bhuvi Ghosh **Sap ID:** 60009210191 **Batch:** D22

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.

Department of Computer Science and Engineering (Data Science)

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)
count ← −1
path ← empty list
depthBound ← 0
repeat
    previousCount ← count
    (count, path) ← DB-DFS-2(S, depthBound)
    depthBound ← depthBound + 1
until (path is not empty) or (previousCount = count)
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.
count ← 0
OPEN ← (S, null, 0) : []
CLOSED ← empty list
while OPEN is not empty
    nodePair ← head OPEN
    (N, _, depth) ← nodePair
    if GoalTest(N) = TRUE
        return (count, ReconstructPath(nodePair, CLOSED))
    else CLOSED ← nodePair : CLOSED
        if depth < depthBound
            neighbours ← MoveGen(N)
            newNodes ← RemoveSeen(neighbours, OPEN, [] )
            newPairs ← MakePairs(newNodes, N, depth + 1)
            OPEN ← newPairs ++ tail OPEN
            count ← count + length newPairs
        else OPEN ← tail OPEN
return (count, empty list)
```

Auxiliary Functions for DFID

```
MAKEPAIRS(nodeList, parent, depth)
if nodeList is empty
    return empty list
else nodePair ← (head nodeList, parent, depth)
    return nodePair : MAKEPAIRS(tail nodeList, parent, depth)
```

```
RECONSTRUCTPATH(nodePair, CLOSED)
SKIPTO(parent, nodePairs, depth)
    if (parent, _, depth) = head nodePairs
        return nodePairs
    else return SKIPTO(parent, tail nodePairs, depth)

(node, parent, depth) ← nodePair
path ← node : []
while parent is not null
    path ← parent : path
    CLOSED ← SKIPTO(parent, CLOSED, depth – 1)
    (_, parent, depth) ← head CLOSED
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 3: Graph

```
import itertools
import math
import sys

def distance(point1, point2):
    x1, y1 = point1
    x2, y2 = point2
    return math.sqrt((x1 - x2) ** 2 + (y1 - y2) ** 2)


def path_length(path, cities):
    total_distance = 0
    for i in range(len(path) - 1):
```

```
        total_distance += distance(cities[path[i]], cities[path[i +
1]])
    return total_distance

def check_goal_test(path, num_cities):
    return len(path) == num_cities + 1

def dfs_tsp(cities, current_city, depth, path, best_path,
best_length):
    num_cities = len(cities)

    if depth == num_cities:
        length = path_length(path, cities)
        if length < best_length[0]:
            best_length[0] = length
            best_path[0] = path.copy()
        return

    for next_city in range(num_cities):
        if next_city not in path:
            path.append(next_city)
            dfs_tsp(cities, next_city, depth + 1, path, best_path,
best_length)
            path.pop()

def dfid_tsp(cities):
    num_cities = len(cities)
    start_city = 0
    best_path = [None]
    best_length = [sys.float_info.max]

    for depth_limit in range(1, num_cities):
        dfs_tsp(cities, start_city, 1, [start_city], best_path,
best_length)

    return best_path[0], best_length[0]

cities = [(0, 0), (1, 2), (2, 4), (3, 1)]
best_path, best_length = dfid_tsp(cities)
print("Best Length:",
best_length+distance(cities[0],cities[best_path[-1]]))
```

```
best_path.append(0)
print("Best Path:", best_path)
```

Output:

```
Best Length: 10.79669127533634
Best Path: [0, 1, 2, 3, 0]
```