

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.

- 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 \leftarrow -1
2 path ← empty list
3 depthBound \leftarrow 0
   repeat
5
        previousCount ← count
6
        (count, path) \leftarrow DB-DFS-2(S, depthBound)
7
        depthBound \leftarrow 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.
10 count \leftarrow 0
11 OPEN \leftarrow (S, null, 0): []
12 CLOSED ← empty list
    while OPEN is not empty
13
         nodePair ← head OPEN
14
15
         (N, \underline{\hspace{1em}}, depth) \leftarrow nodePair
16
         if GOALTEST(N) = TRUE
17
              return (count, RECONSTRUCTPATH(nodePair, CLOSED))
18
         else CLOSED ← nodePair : CLOSED
19
              if depth < depthBound
20
                   neighbours \leftarrow MOVEGEN(N)
21
                   newNodes ← REMOVESEEN(neighbours, OPEN, [])
22
                   newPairs \leftarrow MAKEPAIRS(newNodes, N, depth + 1)
23
                   OPEN ← newPairs ++ tail OPEN
24
                   count \leftarrow count + length newPairs
25
              else OPEN \leftarrow 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)
        return nodePair: MAKEPAIRS(tail nodeList, parent, depth)
RECONSTRUCTPATH(nodePair, CLOSED)
 1 SKIPTo(parent, nodePairs, depth)
         if (parent, \underline{\phantom{a}}, 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
         path ← parent: path
 9
         CLOSED \leftarrow SKIPTo(parent, CLOSED, depth - 1)
         (_, parent, depth) ← head CLOSED
10
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 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):
```

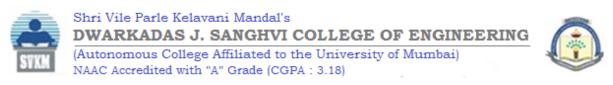
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING



(Autonomous College Affiliated to the University of Mumbai) NAAC Accredited with "A" Grade (CGPA: 3.18)

Department of Computer Science and Engineering (Data Science)

```
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):
    if depth == num cities:
        length = path length(path, cities)
        if length < best length[0]:</pre>
            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)
   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]