



Department of Computer Science and Engineering (Data Science)

Subject: Artificial Intelligence (DJ19DSC502)

AY: 2023-24

Name-Jhanvi Parekh

Sap id-60009210033

Batch-D11

Experiment 2

(Uninformed Search)

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 stateof-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 (statetransition 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



Shri Vile Parle Kelavani Mandal's

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)

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



Department of Computer Science and Engineering (Data Science)

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



Department of Computer Science and Engineering (Data Science)

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



Department of Computer Science and Engineering (Data Science)

```
class Graph:
    def __init__(self):
        self.graph = {}

    def add_edge(self, u, v):
        if u not in self.graph:
            self.graph[u] = []
        self.graph[u].append(v)

    def move_generation(self, node):
        if node in self.graph:
            return self.graph[node]
        else:
            return []

    def goal_test(self, node, goal):
        return node == goal

    def find_path(self, start, goal):
        visited = set()
        stack = [[start]]

        while stack:
            path = stack.pop()
            node = path[-1]

            if node == goal:
                return path

            if node not in visited:
                visited.add(node)
                neighbors = self.move_generation(node)
                for neighbor in neighbors:
                    new_path = list(path)
                    new_path.append(neighbor)
                    stack.append(new_path)

        return None

    def make_pair(self, node1, node2):
        return (node1, node2)

graph = Graph()
graph.add_edge('A', 'B')
graph.add_edge('A', 'C')
graph.add_edge('B', 'D')
graph.add_edge('B', 'E')
graph.add_edge('C', 'F')
graph.add_edge('C', 'G')
graph.add_edge('E', 'I')
graph.add_edge('D', 'H')
graph.add_edge('F', 'H')
graph.add_edge('G', 'H')
graph.add_edge('H', 'I')

start_node = 'A'
goal_node = 'H'

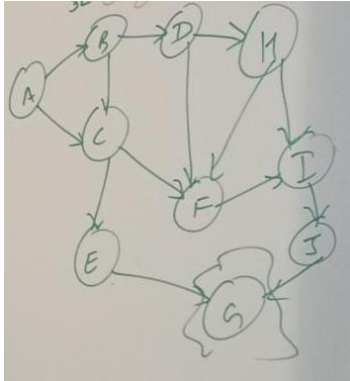
path = graph.find_path(start_node, goal_node)
if path:
    print(f"Path from {start_node} to {goal_node}: {path}")
else:
    print(f"No path found from {start_node} to {goal_node}")
```

Path from A to H: ['A', 'C', 'F', 'H']



Department of Computer Science and Engineering (Data Science)

For the graph given in the lab





Department of Computer Science and Engineering (Data Science)

The screenshot shows a Jupyter Notebook titled "60009210033_JhanviParekh_AI_Lab_2.ipynb". The code implements a Depth-First Search (DFS) algorithm to find a path from node 'A' to node 'H' in a graph. The graph is defined with 10 vertices and edges between 'A' and 'B', 'A' and 'C', and 'B' and 'C'. The DFS function is recursive, exploring all possible paths from the source node 'A' to the target node 'H'. The output of the DFS function is a list of paths, which is printed at the bottom of the notebook.

```
from collections import defaultdict

class Graph:
    def __init__(self, vertices):
        self.v = vertices
        self.graph = defaultdict(list)

    def addEdge(self, u, v):
        self.graph[u].append(v)

    def DFS(self, src, target, maxDepth, path):
        if src == target:
            path.append(src)
            return True

        if maxDepth <= 0:
            return False

        path.append(src)
        for i in self.graph[src]:
            if self.DFS(i, target, maxDepth - 1, path):
                return True

        path.pop()
        return False

    def DFS(self, src, target, maxDepth):
        path = []
        for i in range(maxDepth):
            path.clear()
            if self.DFS(src, target, i + 1, path):
                return path + [target]
        return []

g = Graph(10)
g.addEdge('A', 'B')
g.addEdge('A', 'C')
g.addEdge('B', 'C')
g.addEdge('C', 'H')
```



Department of Computer Science and Engineering (Data Science)

```
path.pop()
return False

def DFS(self, src, target, maxDepth):
    path = []
    for i in range(maxDepth):
        path.clear()
        if self.DFS(src, target, i + 1, path):
            return path + [target]
    return []

g = Graph(10)
g.addEdge('A', 'B')
g.addEdge('A', 'C')
g.addEdge('B', 'C')
g.addEdge('C', 'F')
g.addEdge('D', 'F')
g.addEdge('D', 'H')
g.addEdge('F', 'I')
g.addEdge('H', 'I')
g.addEdge('I', 'J')
g.addEdge('I', 'G')
g.addEdge('J', 'G')

target = 'G'
maxDepth = 4
src = 'A'

path = g.DFS(src, target, maxDepth)

if path:
    print("Target is reachable from source within max depth.")
    print("Path: ", " -> ".join(map(str, path[:-1])))
else:
    print("Target is NOT reachable from source within max depth.")
```

Target is reachable from source within max depth.
Path: A -> C -> E -> G

link:

https://colab.research.google.com/drive/1kVt8EK9R_LXjc5U7yY922AhRuAshg_M?usp=sharing

Analysis:

Time Complexity: DFS explores the search space incrementally, increasing the depth limit in each iteration. In the worst case, it explores all nodes up to the maximum depth d , so the time complexity is $O(b^d)$, where b is the branching factor, and d is the depth of the shallowest solution. In the worst case, it's still exponential.

Space Complexity: The space complexity is $O(bd)$ because it needs to store the search tree up to the current depth limit d . It's linear with respect to the depth. **Optimality:** DFS is optimal if the cost of each step is the same. However, in some cases, it may not be the most efficient algorithm to find the shortest path due to its exponential time complexity.

Completeness: DFS is complete if the branching factor is finite, and the search tree is finite. It will find a solution if one exists within the depth limit. However, if the depth limit is too small, it may not find a solution even if one exists