Chandigarh University
Master of Engineeing—Artificial Intelligence

Experiment 1

Student name: *Aditya Kumar (24MAI14003)*

Subject: *Artificial Intelligence Lab (24CSH-621)*
– Instructor: *Manjit Singh*
Date: *01 Aug, 2024*

Aim: Implement Uninformed Search Strategies: BFS, DFS and ID (Iterative Deepening) Ideas of Game Searching (Chess, Checkers, Tic Tac Toe, Puzzle Game, etc).

1 Task to be done

To implement uninformed search strategies:

1. BFS (Breadth-First Search)
2. DFS (Depth-First Search)
3. ID (Iterative Deepening)

2 Theory

BFS (Breadth-First Search) – BFS is a fundamental algorithm used to explore or search through graphs or tree structures.

DFS (Depth-First Search) – DFS is another fundamental algorithm used to explore or search through graphs or tree structures. Unlike Breadth-First Search (BFS), which explores level by level, DFS explores as far down a branch as possible before backtracking.

ID (Iterative Deepening) – ID combines the depth-first search (DFS) strategy with a breadth-first approach, effectively balancing the benefits of both. It's particularly useful in scenarios where the depth of the solution is not known in advance, such as in certain search problems and puzzles.

3 Algorithm

1. **Algorithm for Breadth-First Search (BFS):**
   (a) Initialization:
   - Create an empty queue
   - Create an empty set or list to keep track of visited nodes
   - Enqueue the starting node and mark it as visited

   (b) Process:
   - While the queue is not empty
     i. Dequeue a node from the front of the queue

ii. Process the node
iii. For each neighnor of the node
- If the neighbor has not been visited
  A. Mark the neighbor as visited
  B. Enqueue the neighbor

2. **Algorithm for Depth-First Search**

(a) Initialization
- Create a set or list to keep track of visited nodes
- Call the recursive DFS function starting from the root node

(b) Process (Recursive DFS)
- If the node has not been visited
  i. Mark the node as visited
  ii. Process the node (e.g., print it, check for a condition, etc.)
  iii. For each neighbor of the node
  - If the neighbor has not been visited
    * Recursively call DFS on the neighbor

3. **Algorithm for Iterative Deepening (ID)**

(a) Initialization
- For each depth from 0 to a specified maximum depth
- Call the depth-limited DFS function with the current depth

(b) Depth-Limited DFS
- If the current depth is 0 and the node has not been visited
  i. Mark the node as visited
  ii. Process the node
- If the current depth is greater than 0
  i. Mark the node as visited
  ii. For each neighbor of the node
  - If the neighbor has not been visited
    * Recursively call depth-limited DFS on the neighbor with depth reduced by 1

4 Pseudocode

1. **Breadth-First Search (BFS)**

```
BFS(graph, start_node):
    Initialize an empty queue
    Initialize a set for visited nodes
    Enqueue the start_node and mark it as visited

    while queue is not empty:
```

```
        node = dequeue from queue
        process the node

        for each neighbor of node:
            if neighbor is not visited:
                mark neighbor as visited
                enqueue neighbor
```

2. **Depth-First Search (DFS)**

```
DFS(graph, node, visited):
    if node is not visited:
        mark node as visited
        process the node

        for each neighbor of node:
            if neighbor is not visited:
                DFS(graph, neighbor, visited)
```

3. **Iterative Deepening**

```
IDDFS(graph, start, max_depth):
    for depth from 0 to max_depth:
        visited = set()
        DFS_Limited(graph, start, depth, visited)

DFS_Limited(graph, node, depth, visited):
    if depth is 0 and node is not visited:
        mark node as visited
        process the node
    elif depth > 0:
        mark node as visited
        for each neighbor of node:
            if neighbor is not visited:
                DFS_Limited(graph, neighbor, depth-1, visited)
```

5 Code

1. **Breadth-First Search**

```
graph = {
    "A": ["B", "C", "D"],
    "B": ["A"],
    "C": ["A", "D"],
    "D": ["A", "C", "E"],
    "E": ["D"],
}


def bfs(node):
    visited = [False] * (len(graph))
```

```
    queue = []
    visited.append(node)
    queue.append(node)
    while queue:
        v = queue.pop(0)
        print(v, end="␣")
        for neigh in graph[v]:
            if neigh not in visited:
                visited.append(neigh)
                queue.append(neigh)


if __name__ == "__main__":
    bfs("A")
```

2. Depth-First Search

```
from collections import defaultdict


class Graph:
    def __init__(self):
        self.adj = defaultdict(list)

    def insertEdge(self, u, v):
        self.adj[u].append(v)
        self.adj[v].append(u)

    def DFS_helper(self, u, visited):
        visited.add(u)
        print(u, end="␣")
        for v in self.adj[u]:
            if v not in visited:
                self.DFS_helper(v, visited)

    def DFS(self, u):
        visited = set()
        self.DFS_helper(u, visited)


g = Graph()
g.insertEdge(0, 1)
g.insertEdge(0, 3)
g.insertEdge(1, 4)
g.insertEdge(1, 2)
g.insertEdge(2, 3)
g.insertEdge(4, 5)
g.insertEdge(4, 6)
g.insertEdge(5, 6)
print("The␣DFS␣traversal␣of␣the␣given␣graph␣starting␣from␣node␣0␣is␣-")
g.DFS(0)
```

3. Iterative Deepening

```
class Node(object):
    def __init__(self, label: str = None):
        self.label = label
        self.children = []
```

```
5
6       def __lt__(self, other):
7           return self.label < other.label
8
9       def __gt__(self, other):
10          return self.label > other.label
11
12      def __repr__(self):
13          return "{} -> {}".format(self.label, self.children)
14
15      def add_child(self, node, cost=1):
16          if type(node) is list:
17              [self.add_child(sub_node) for sub_node in node]
18              return
19          edge = Edge(self, node, cost)
20          self.children.append(edge)
21
22
23  class Edge(object):
24      def __init__(
25          self,
26          source: Node,
27          destination: Node,
28          cost: int = 1,
29          bidirectional: bool = False,
30      ):
31          self.source = source
32          self.destination = destination
33          self.cost = cost
34          self.bidirectional = bidirectional
35
36      def __repr__(self):
37          return "{}: {}".format(self.cost, self.destination.label)
38
39
40  A = Node("A")
41  B = Node("B")
42  C = Node("C")
43  D = Node("D")
44  E = Node("E")
45  F = Node("F")
46  G = Node("G")
47  A.add_child([B, C, E])
48  B.add_child([A, D, F])
49  C.add_child([G, A])
50  D.add_child(B)
51  E.add_child([F, A])
52  F.add_child([E, B])
53  G.add_child(C)
54  _ = [print(node) for node in [A, B, C, D, E, F, G]]
55
56
57  def iddfs(root: Node, goal: str, maximum_depth: int = 10):
58      for depth in range(0, maximum_depth):
59          result = _dls([root], goal, depth)
60          if result is None:
61              continue
62          return result
```

```
63      raise ValueError("goal␣not␣in␣graph␣with␣depth␣{}".format(maximum_depth))
64
65
66  def _dls(path: list, goal: str, depth: int):
67      current = path[-1]
68      if current.label == goal:
69          return path
70      if depth <= 0:
71          return None
72      for edge in current.children:
73          new_path = list(path)
74          new_path.append(edge.destination)
75          result = _dls(new_path, goal, depth - 1)
76          if result is not None:
77              return result
78
79
80  iddfs(D, "G")
```

6 Output

1. **Breadth-First Search**

```
~/dev/ai via 🐍 v3.12.4 via ❄  impure (nix-shell-env)
❯ python e1-bfs.py
A B C D E %
```

2. **Depth-First Search**

```
~/dev/ai via 🐍 v3.12.4 via ❄  impure (nix-shell-env)
❯ python e1-dfs.py
The DFS traversal of the given graph starting from node 0 is -
0 1 4 5 6 2 3 %
```

3. **Iterative Deepening**

```
~/dev/ai via 🐍 v3.12.4 via ❄  impure (nix-shell-env)
❯ python e1-id.py
A → [1: B, 1: C, 1: E]
B → [1: A, 1: D, 1: F]
C → [1: G, 1: A]
D → [1: B]
E → [1: F, 1: A]
F → [1: E, 1: B]
G → [1: C]
```

7 Learning Outcomes

1. Learned about BFS
2. Learned about DFS
3. Learned about ID