ICT4133 – Artificial Intelligence

Practical 06: Search Strategies and Propositional Logic

Part A – Uninformed Search Algorithms

1. Breadth-First Search (BFS)

Breadth-First Search explores nodes level by level using a **queue (FIFO)** structure.

Properties

- Complete: Yes
- Optimal: Yes (for unweighted graphs)
- Time Complexity: **O(V + E)**
- Space Complexity: **O(V)**

Python Implementation

```
from collections import deque

def bfs(graph, start):
    visited = set()
    queue = deque([start])
    visited.add(start)

    while queue:
        node = queue.popleft()
        print(node, end=" ")

        for neighbor in graph[node]:
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append(neighbor)

# Example Graph
graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F'],
    'D': [],
    'E': [],
    'F': []
}

print("BFS Traversal:")
bfs(graph, 'A')
```

Lab Task 01

1. Modify BFS to return the shortest path between two nodes.
2. Apply BFS on a 2D grid maze.

2. Depth-First Search (DFS)

DFS explores deeply before backtracking using **recursion or stack (LIFO)**.

Properties

- Complete: No (in infinite-depth spaces)
- Optimal: No
- Time Complexity: **O(V + E)**
- Space Complexity: **O(V)**

Recursive Implementation

```
def dfs(graph, start, visited=None):
    if visited is None:
        visited = set()

    visited.add(start)
    print(start, end=" ")

    for neighbor in graph[start]:
        if neighbor not in visited:
            dfs(graph, neighbor, visited)

# Example
print("DFS Traversal:")
dfs(graph, 'A')
```

Iterative DFS Version

```
def dfs_iterative(graph, start):
    visited = set()
    stack = [start]

    while stack:
        node = stack.pop()
        if node not in visited:
            print(node, end=" ")
            visited.add(node)
            stack.extend(reversed(graph[node]))
```

Lab Task 02

- Compare BFS and DFS traversal outputs.
- Write a program to detect cycles using DFS.

Part B – Informed Search Algorithms

3. Greedy Best-First Search

Uses heuristic function **h(n)**.
Selects node with lowest heuristic value.

Python Implementation

```
import heapq

def greedy_best_first(graph, start, goal, heuristic):
    visited = set()
    priority_queue = [(heuristic[start], start)]

    while priority_queue:
        _, current = heapq.heappop(priority_queue)

        if current == goal:
            print("Goal reached:", current)
            return

        if current not in visited:
            visited.add(current)
            print(current, end=" ")

            for neighbor in graph[current]:
                if neighbor not in visited:
                    heapq.heappush(priority_queue, (heuristic[neighbor],
neighbor))
```

4. A Search Algorithm*

Evaluation function:

f(n)=g(n)+h(n)

Where:

- g(n) = cost so far
- h(n) = heuristic

A Implementation*

```
import heapq

def a_star(graph, start, goal, heuristic):
    open_list = []
    heapq.heappush(open_list, (0, start))

    g_cost = {start: 0}
    parent = {start: None}

    while open_list:
        _, current = heapq.heappop(open_list)
```

```
        if current == goal:
            path = []
            while current:
                path.append(current)
                current = parent[current]
            return path[::-1]

        for neighbor, cost in graph[current]:
            new_cost = g_cost[current] + cost

            if neighbor not in g_cost or new_cost < g_cost[neighbor]:
                g_cost[neighbor] = new_cost
                f_cost = new_cost + heuristic[neighbor]
                heapq.heappush(open_list, (f_cost, neighbor))
                parent[neighbor] = current
```

Lab Task 03

- Implement A* for 8-puzzle.
- Compare Greedy vs A* in terms of optimality.

Part C – Propositional Logic in Python

5. Logical Connectives Implementation

Negation

```
def negation(p):
    return not p
```

Conjunction

```
def conjunction(p, q):
    return p and q
```

Disjunction

```
def disjunction(p, q):
    return p or q
```

Exclusive OR

```
def xor(p, q):
    return (p and not q) or (not p and q)
```

Implication

```
def implication(p, q):
    return (not p) or q
```

Bi-Implication

```
def biconditional(p, q):
    return p == q
```

Truth Table Generator

```
from itertools import product

def truth_table():
    print("p  q  p→q")
    for p, q in product([True, False], repeat=2):
        print(p, q, implication(p, q))

truth_table()
```

Lab Task 04

1. Generate truth table for:
    - (p ∨ ¬q) ∧ ¬p
2. Implement a simple model checker.
3. Convert logical expressions to CNF form.
4. Implement:
    - Uniform Cost Search
    - Iterative Deepening DFS
5. Compare all search strategies experimentally.
6. Implement propositional logic inference engine.

7. Write a Python function that:

- Takes a graph (adjacency list)
- Takes `start` and `goal`
- Returns the **shortest path** using BFS (not just traversal)

Output format example:

```
['A', 'C', 'F', 'G']
```

8. Given a 2D grid:

```
grid = [
    [0, 1, 0, 0],
    [0, 0, 0, 1],
    [1, 0, 1, 0],
    [0, 0, 0, 0]
]
```

Where:

- `0 = free cell`
- `1 = blocked`

9. Write a BFS program to:

    - Find shortest path from `(0,0)` to `(3,3)`
    - Print path length

10. Write a DFS-based function that returns:

```
True  # if cycle exists
False # otherwise
```

    Hint: Use recursion stack.

11. Implement Topological Sort using DFS for a Directed Acyclic Graph (DAG).

    Output:

```
['A', 'C', 'B', 'D']
```

12. Write a program that:

    - Generates all possible next states of 8-puzzle
    - Avoid revisiting previous states

13. Write a program that:

    - Runs BFS, DFS, UCS, A*
    - On same graph
    - Prints:
        - Path
        - Cost
        - Nodes expanded
        - Time taken

14. A game tree represents possible moves in a game, where:

    - Each node represents a game state
    - The root node represents the initial state
    - Leaf nodes represent terminal states
    - Edges represent possible moves

    Write a Python program to:

    1. Represent a game tree using an adjacency list (dictionary format).
    2. Implement **Depth-First Search (DFS)**.
    3. Traverse the game tree starting from the root node.
    4. Print **all possible root-to-leaf paths**.

15. A robot is placed in a 2D grid environment where some cells contain obstacles. The robot can move only in four directions:

    - Up
    - Down
    - Left
    - Right

    Diagonal movement is **not allowed**.

    Each cell in the grid is represented as:

    - `0` → Free cell
    - `1` → Obstacle

    The robot starts from a given **start position** and must reach a given **goal position**.

    Write a complete Python program to:

    1. Implement the **A* Search Algorithm**.
    2. Use **Manhattan distance** as the heuristic function.
    3. Avoid diagonal movements.
    4. Find the shortest path from the start position to the goal position.
    5. Print the final path visually in the grid using:
        - `S` → Start
        - `G` → Goal
        - `*` → Path
        - `1` → Obstacles
        - `0` → Unvisited free cells

16. Write a Python program to generate the truth table for:

    $(p \wedge q) \rightarrow r$

17. Write a function to compute and print the truth table for:

    $(p \vee q) \wedge (\neg r \vee p)$

18. Generate a truth table for:

    $(p \oplus q) \leftrightarrow r$

    Implement XOR manually without using built-in operators.

19. Generate the truth table for:

    $(p \wedge \neg q) \vee (r \wedge \neg p)$

20. Write a program to check whether the following two propositions are logically equivalent:

    1. $p \rightarrow q$
    2. $\neg p \vee q$

    Your program must:

    - Generate truth tables
    - Compare final columns
    - Print: "Logically Equivalent" or "Not Equivalent"

21. Write a Python program to determine whether the following expression is a tautology:

    $(p \rightarrow q) \vee (q \rightarrow p)$

22. Write a program to check if the following is always false:

    $(p \wedge \neg p)$

23. Write a function that determines whether an expression is:

    - Tautology
    - Contradiction
    - Contingency

    Test it using:

    $(p \vee q) \wedge (\neg p \vee \neg q)$

24. Given:

    1. $p \rightarrow q$
    2. $q \rightarrow r$
    3. p

    Write a program to:

    - Determine if r is logically entailed.

25. Create a simple model-checking function that:

    - Accepts a knowledge base (list of expressions)
    - Accepts a query
    - Returns True if KB entails query

26. Write a Python program that:

    - Takes logical expression as input (e.g., p and not q)
    - Automatically generates truth table
    - Evaluates expression

27. Write a function that converts:

    $p \rightarrow (q \wedge r)$

    into its equivalent CNF form programmatically.

28. Write a program that:

    - Automatically detects number of variables in expression
    - Generates complete truth table dynamically