**Assignment 01**

**AI LAB**

**MISBAH UR REHMAN SAIM**

**SP24-BSE-036**

## 1) Depth-First Search (DFS)

|  |  |
| --- | --- |
| **OPEN** | **CLOSED** |
| Arad | [] |
| Timisoara, Sibiu, Zerind | [Arad] |
| Timisoara, Sibiu, Oradea | [Arad, Zerind] |
| Timisoara, Sibiu, Sibiu | [Arad, Zerind, Oradea] |
| Timisoara, Sibiu, Rimnicu Vilcea, Fagaras | [Arad, Zerind, Oradea, Sibiu] |
| Timisoara, Sibiu, Rimnicu Vilcea, Bucharest | [Arad, Zerind, Oradea, Sibiu, Fagaras, **Bucharest**] |

## 2) Breadth-First Search (BFS) with OPEN and CLOSED

| **OPEN** | **CLOSED** |
| --- | --- |
| [Arad] | [] |
| [Zerind, Sibiu, Timisoara] | [Arad] |
| [Sibiu, Timisoara, Oradea] | [Arad, Zerind] |
| [Timisoara, Oradea, Rimnicu Vilcea, Fagaras] | [Arad, Zerind, Sibiu] |
| [Oradea, Rimnicu Vilcea, Fagaras, Lugoj] | [Arad, Zerind, Sibiu, Timisoara] |
| [Rimnicu Vilcea, Fagaras, Lugoj] | [Arad, Zerind, Sibiu, Timisoara, Oradea] |
| [Fagaras, Lugoj, Pitesti, Craiova] | [Arad, Zerind, Sibiu, Timisoara, Oradea, Rimnicu Vilcea] |
| [Lugoj, Pitesti, Craiova, Bucharest] | [Arad, Zerind, Sibiu, Timisoara, Oradea, Rimnicu Vilcea, Fagaras, Bucharest] |

## 3) Python Implementation

**BFS**

## from collections import deque

## def breadth\_first\_search(graph, start, goal):

## # Initialize

## open\_list = deque([start]) # queue for nodes to explore

## closed\_list = [] # visited nodes

## while open\_list: # while states remain

## # Remove leftmost state (FIFO)

## X = open\_list.popleft()

## print(f"Visiting: {X}")

## print(f"OPEN: {list(open\_list)} \t CLOSED: {closed\_list}")

## # Goal test

## if X == goal:

## print(f"SUCCESS: Goal '{goal}' found!")

## return True

## # Generate children

## children = graph.get(X, [])

## # Mark current node as visited

## closed\_list.append(X)

## # Add children (if not already in OPEN or CLOSED)

## for child in children:

## if child not in open\_list and child not in closed\_list:

## open\_list.append(child)

## # If no states left

## print("FAIL: Goal not found.")

## return False

## # Romania map (partial graph for Arad → Bucharest search)

## graph = {

## "Arad": ["Zerind", "Sibiu", "Timisoara"],

## "Zerind": ["Arad", "Oradea"],

## "Oradea": ["Zerind", "Sibiu"],

## "Sibiu": ["Arad", "Oradea", "Fagaras", "Rimnicu Vilcea"],

## "Fagaras": ["Sibiu", "Bucharest"],

## "Rimnicu Vilcea": ["Sibiu", "Pitesti", "Craiova"],

## "Pitesti": ["Rimnicu Vilcea", "Craiova", "Bucharest"],

## "Bucharest": ["Fagaras", "Pitesti", "Giurgiu", "Urziceni"],

## "Timisoara": ["Arad", "Lugoj"],

## "Lugoj": ["Timisoara", "Mehadia"],

## "Mehadia": ["Lugoj", "Drobeta"],

## "Drobeta": ["Mehadia", "Craiova"],

## "Craiova": ["Drobeta", "Rimnicu Vilcea", "Pitesti"],

## "Giurgiu": ["Bucharest"],

## "Urziceni": ["Bucharest", "Vaslui", "Hirsova"],

## "Hirsova": ["Urziceni", "Eforie"],

## "Eforie": ["Hirsova"],

## "Vaslui": ["Iasi", "Urziceni"],

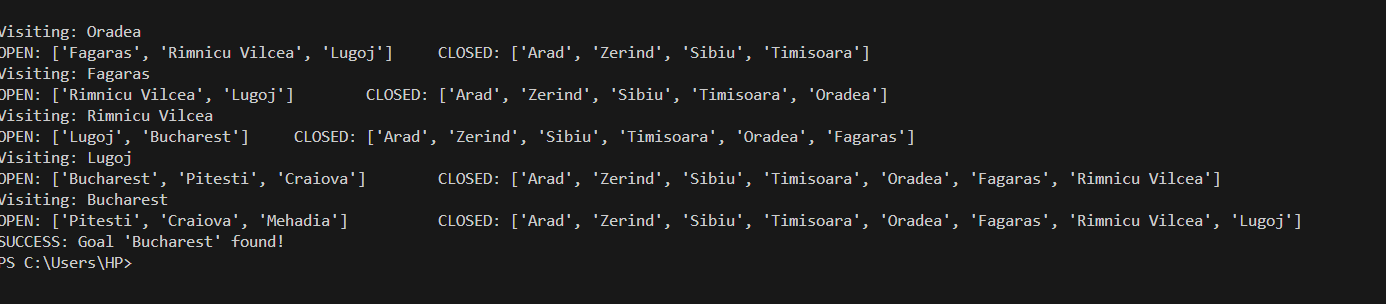
## "Iasi": ["Neamt", "Vaslui"],

## "Neamt": ["Iasi"]

## }

## # Run BFS

## breadth\_first\_search(graph, "Arad", "Bucharest")



**DFS**

def depth\_first\_search(graph, start, goal):

# Initialize

open\_list = [start] # stack for nodes to explore (LIFO)

closed\_list = [] # visited nodes

while open\_list: # while states remain

# Remove rightmost state (LIFO stack)

X = open\_list.pop()

print(f"Visiting: {X}")

print(f"OPEN: {open\_list} \t CLOSED: {closed\_list}")

# Goal test

if X == goal:

print(f"SUCCESS: Goal '{goal}' found!")

return True

# Generate children

children = graph.get(X, [])

# Mark current node as visited

closed\_list.append(X)

# Push children (if not already in OPEN or CLOSED)

# reversed() ensures left-to-right expansion order

for child in reversed(children):

if child not in open\_list and child not in closed\_list:

open\_list.append(child)

# If no states left

print("FAIL: Goal not found.")

return False

# Romania map (same as BFS)

graph = {

"Arad": ["Zerind", "Sibiu", "Timisoara"],

"Zerind": ["Arad", "Oradea"],

"Oradea": ["Zerind", "Sibiu"],

"Sibiu": ["Arad", "Oradea", "Fagaras", "Rimnicu Vilcea"],

"Fagaras": ["Sibiu", "Bucharest"],

"Rimnicu Vilcea": ["Sibiu", "Pitesti", "Craiova"],

"Pitesti": ["Rimnicu Vilcea", "Craiova", "Bucharest"],

"Bucharest": ["Fagaras", "Pitesti", "Giurgiu", "Urziceni"],

"Timisoara": ["Arad", "Lugoj"],

"Lugoj": ["Timisoara", "Mehadia"],

"Mehadia": ["Lugoj", "Drobeta"],

"Drobeta": ["Mehadia", "Craiova"],

"Craiova": ["Drobeta", "Rimnicu Vilcea", "Pitesti"],

"Giurgiu": ["Bucharest"],

"Urziceni": ["Bucharest", "Vaslui", "Hirsova"],

"Hirsova": ["Urziceni", "Eforie"],

"Eforie": ["Hirsova"],

"Vaslui": ["Iasi", "Urziceni"],

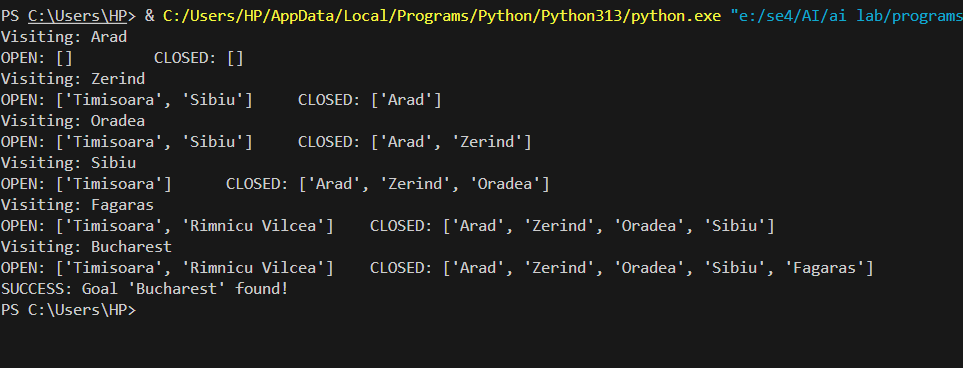
"Iasi": ["Neamt", "Vaslui"],

"Neamt": ["Iasi"]

}

# Run DFS

depth\_first\_search(graph, "Arad", "Bucharest")



## 3) Comparison and Analysis

DFS explores one branch deeply before backtracking. It finds a path quickly but does not guarantee the shortest path. In our run, DFS path had 6 cities.  
  
BFS explores level by level using a queue. The first time it finds the goal, that path is guaranteed shortest in terms of number of cities. In our run, BFS path had 4 cities, which is optimal.  
  
Thus, BFS performs better than DFS for shortest paths in unweighted graphs, because it systematically explores all nodes at smaller depth before going deeper.