Problem Set: BFS and DFS

Prof. Rodrigo Silva

Introduction

This problem set breaks down the implementation of BFS (Breadth-First Search) and DFS (Depth-First Search) into smaller parts. The goal is to help students gradually implement each component and then combine them to form the full algorithms.

Part 1: Representing the Graph

Problem 1.1: Graph Representation

Write a function to represent a graph using a dictionary, where each key is a node and the value is a list of its neighbors. (See `https://www.w3schools.com/python/python_dictionaries.asp` to learn how to use dictionaries (the data structure used to represent the graph) in Python).

```
Graph = {
    'A': ['B', 'C'],
    'B': ['A', 'D'],
    'C': ['A', 'E'],
    'D': ['B'],
    'E': ['C']
}
```

The function `add_edge(node1, node2)` should update this dictionary representation.

Algorithm 1 add_edge(node1, node2)

```
if node1 is not in Graph then
  Graph[node1] ← empty list
end if
if node2 is not in Graph then
  Graph[node2] ← empty list
end if
append node2 to Graph[node1]
append node1 to Graph[node2] {since the graph is undirected}
```

—

Part 2: BFS - Breadth-First Search

Problem 2.1: Initializing the Queue

Write a function that initializes a queue for BFS using a Python list.

Algorithm 2 Initialize_Queue(start_node)

```
queue ← [start_node] {Initialize queue with start_node}
return queue
```

—

Problem 2.2: Handling Neighbors

Write a function to visit a node and enqueue all its unvisited neighbors.

Algorithm 3 Visit_Neighbors(graph, node, visited, queue)

```
for each neighbor in graph[node] do
  if neighbor is not in visited then
    queue.append(neighbor) {enqueue neighbor}
    visited.add(neighbor)
  end if
end for
```

—

Problem 2.3: BFS Complete

Now, put everything together to complete the BFS algorithm.

Algorithm 4 BFS(graph, start_node)

```
visited ← set() {empty set}
queue ← Initialize_Queue(start_node)
visited.add(start_node)
while queue is not empty do
  node ← queue.pop(0) {dequeue from front}
  print "Visited", node
  Visit_Neighbors(graph, node, visited, queue)
end while
```

—

Part 3: DFS - Depth-First Search

Problem 3.1: Initializing the Stack

Write a function that initializes a stack for DFS using a Python list.

Algorithm 5 Initialize_Stack(start_node)

```
stack ← [start_node] {Initialize stack with start_node}
return stack
```

—

Problem 3.2: Handling Neighbors in DFS

Write a function to visit a node and push all its unvisited neighbors to the stack.

Algorithm 6 Visit_Neighbors_DFS(graph, node, visited, stack)

```
for each neighbor in graph[node] do
  if neighbor is not in visited then
    stack.append(neighbor) {push neighbor onto stack}
    visited.add(neighbor)
  end if
end for
```

—

Problem 3.3: DFS Complete

Now, put everything together to complete the DFS algorithm.

Algorithm 7 DFS(graph, start_node)

```
visited ← set() {empty set}
stack ← Initialize_Stack(start_node)
visited.add(start_node)
while stack is not empty do
  node ← stack.pop() {pop from the stack}
  print "Visited", node
  Visit_Neighbors_DFS(graph, node, visited, stack)
end while
```

—

Part 4: Recursive DFS (Optional)

Problem 4.1: Recursive DFS

Write a recursive function to perform DFS.

Algorithm 8 DFS_Recursive(graph, node, visited)

```
visited.add(node)
print "Visited", node
for each neighbor in graph[node] do
  if neighbor is not in visited then
    DFS_Recursive(graph, neighbor, visited)
  end if
end for
```

—

Final Step: Putting It All Together

At the end, students should test BFS and DFS on a sample graph by combining all the steps. This will help reinforce how each part fits into the whole algorithm.