Lab 4: 9-Tile Problem

Brandon Trinkle

Arizona State University

Course Number: IFT 360

Professor Durgesh Sharma

9/3/24

**Question 1: Understanding the Code**

*Open each of the downloaded files in Spyder. go through the code and write a one sentence explanation for each class, and each method in each class. Specify the class attributes. Go through the classes in this order:*

* *Graph*
* *TreeNode*
* *Tree*
* *GraphTraverser*

*Then check the main.py file to see how the graph representing the map of Romania is constructed.*

**Answer:**

**Graph**

The Graph class is designed to represent a graph structure with nodes and edges. The class contains the following attributes and methods:

* **Attributes**
  + nodes: A dictionary that stores the names of the nodes.
  + edges: A dictionary that stores edges as adjacency lists, mapping nodes to their connected nodes and edge weights.
* **Methods**
  + \_\_init\_\_(self): Initializes the graph with empty dictionaries for nodes and edges.
  + add\_node(self, node\_name): Adds a node to the graph.
  + add\_dir\_edge(self, src\_node, dst\_node, edge\_wt): Adds a directed edge from the source node to the destination node with weight associated with route (wt).
  + add\_bidir\_edge(self, src\_node, dst\_node, edge\_wt): Adds a bidirectional edge between the source and destination nodes with weight associated with route (wt).
  + expand\_node(self, node): Expands a node by returning its list of connected nodes (children).

**TreeNode**

The TreeNode class represents a node in a tree structure. The class contains the following attributes and methods:

* **Attributes:**
  + name: The name of the node.
  + parent: The name of the parent node.
  + children: A dictionary that maps child node names to their edge costs.
* **Methods:**
  + \_\_init\_\_(self, node\_name, parent\_name): Initializes the node with a name, a parent, and an empty dictionary of children.
  + add\_child(self, child\_name, edge\_cost): Adds a child node to the current node’s children with an associated edge cost.
  + set\_parent(self, p): Sets the parent of the node.
  + remove\_child(self, child\_name): Removes a specified child from the node's children dictionary.

**Tree**

The Tree class represents a hierarchical data structure composed of interconnected nodes, used to model relationships such as family trees, organizational charts, or decision processes. This class manages the overall structure of the tree, including node relationships and traversal operations.

* **Attributes:**
  + - nodes: A dictionary where each key is a node name (string), and the value is an instance of the TreeNode class, representing the nodes in the tree.
    - root\_name: A string that holds the name of the root node of the tree.
* **Methods:**
  + - \_\_init\_\_(self): Initializes an empty tree with no nodes and an undefined root.
    - set\_root(self, r): Sets the root node of the tree, linking the node as the root and removing its parent reference.
    - add\_child\_node(self, child\_node, parent\_name, edge\_cost): Adds a child node to the specified parent node within the tree.
    - print\_path(self, n): Prints the path from the root to the specified node, showing the sequence of nodes leading to the target.

**GraphTraverser**

The GraphTraverser class is designed to perform traversal operations on a graph, by searching for paths between nodes.

* **Attributes:**
  + - graph: This attribute stores an instance of the Graph class, representing the graph structure that the traverser will operate on.
    - tree: Used to construct a tree during graph traversal.
  + **Methods:**
    - \_\_init\_\_(self, graph): Initializes the GraphTraverser by taking a Graph object as an input and setting it as the graph to be traversed.
    - breadth\_first\_traversal(self, start\_node, target\_node): This method explores all nodes at the present depth level before moving on to nodes at the next depth level. The method starts at the start\_node and continues until the target\_node is found, or all nodes are explored.

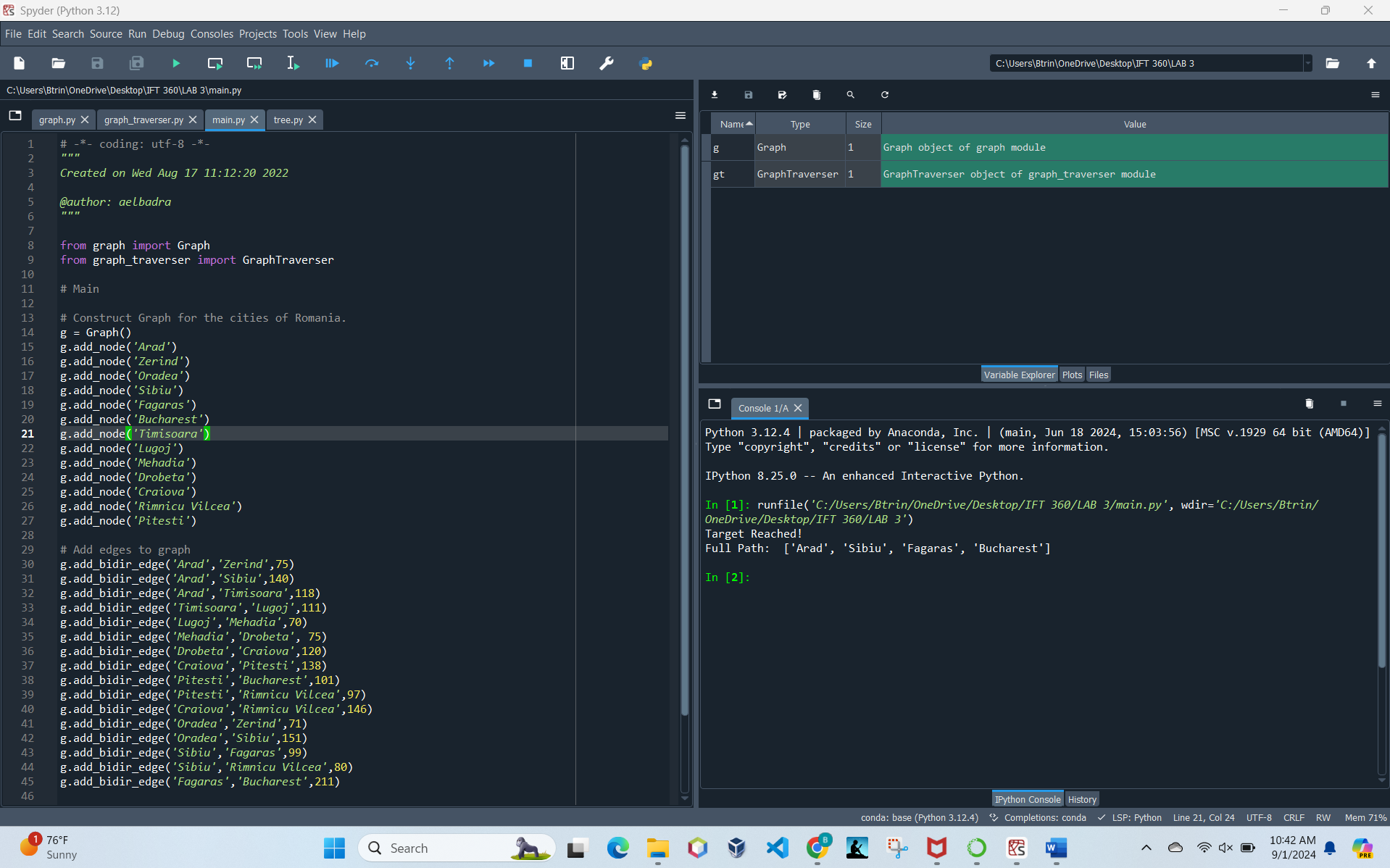
**Main**

The main.py file sets up the graph of Romania using the Graph class by adding nodes for each city and edges to represent the cost or distance between these cities with their respective weights (wt).

**Question 2:**

*Run the main.py file. It should print the path from Arad to Bucharest, as returned by the breadth-first traversal algorithm. What is the printed path? take a screenshot to show that you ran the code and got the path.*

**Answer:**

The output shows the sequence of cities during traversal from the starting point, Arad, to the destination, Bucharest. The GraphTraverser identified the path using the breadth-first search algorithm.

**Question 3:**

*Write a depth-first traversal function based on the discussion above.*

**Answer:**

**Code for Depth-First Search:**

* **GraphTraverser.py**

#Creating depth first traversal

#Created by Brandon Trinkle

#IFT 360

def depth\_first\_traversal(self, start\_node\_name, target\_node\_name):

# Queue for nodes to expand, using last in first out strategy for depth-first search

nodes\_to\_expand = [start\_node\_name]

visited = set()

path = []

while len(nodes\_to\_expand) > 0:

# Popping the last node in the queue for DFS

node\_name = nodes\_to\_expand.pop()

path.append(node\_name)

# If target is found, return the path

# Added some additional print statements for fun

if node\_name == target\_node\_name:

print("Target Reached!")

print(f"Starting point: {start\_node\_name}")

print(f"Destination: {target\_node\_name}")

print(f"Full Path: {path}")

return path

# Skip if already visited

if node\_name in visited:

path.pop() # Remove the node since it's not part of the valid path

continue

visited.add(node\_name)

# Expand the current node and add its children to the stack

for child, \_ in self.graph.expand\_node(node\_name):

if child not in visited:

nodes\_to\_expand.append(child)

print("Target not found!")

return None

A screenshot of a computer

Description automatically generated

* **Main.py**

# Perform breadth-first search to find the path from Arad ==> Bucharest

gt = GraphTraverser(g)

gt.breadth\_first\_traversal('Arad', 'Bucharest')

print("") #Adding break between breadth-first search and depth-first search

# Depth-first traversal from Arad to Bucharest

# Brandon Trinkle

# IFT 360

print("Running Depth-First Traversal:")

gt.depth\_first\_traversal('Arad', 'Bucharest')

A screenshot of a computer

Description automatically generated

**Question 4:**

*Call the function to find a path from Arad to Bucharest. Is the resulting path the same as the breadth-first path?*

**Answer:**

No, the resulting path from the depth-first traversal is not the same as the breadth-first path. Depth-First traversal found more cities. This is due to depth-first exploring the path in its entirety before exploring a new node. Conversely, breadth-first search will search in levels left to right, stopping when it gets to its destination. Breadth-first search will find the shortest route, while depth-first search will find a more detailed route. This demonstrates the differences between the two traversal methodes.

* *Breadth-First Search found a direct, shorter path* [Arad, Sibiu, Fagaras, Bucharest].
* *Depth-First found a longer route* [Arad, Timisoara, Lugoj, Mehadia, Drobeta, Craiova, Rimnicu Vilcea, Sibiu, Fagaras, Bucharest].

A screenshot of a computer

Description automatically generated