CENG310 Programming Assignment 4: Graph Exploration Game

Instructor: Çiğdem Avcı
Teaching Assistant: Ardan Yılmaz

Due Date: 30.05.2024, 23.55

Abstract

This programming assignment focuses on the application of graph algorithms in a game scenario. You will implement a weighted graph data structure and apply Depth-First Search (DFS), Breadth-First Search (BFS), and Dijkstra's algorithm to navigate through a graph-based game world. Through this assignment, you will gain hands-on experience with fundamental graph traversal and pathfinding techniques.

1 Introduction

In this assignment, you will develop a simple game called "Graph Explorer." The game world is represented as a weighted graph, where nodes represent locations and edges represent paths inbetween, with weights indicating the cost to traverse the path.

2 Objective

The primary objective of this assignment is to strengthen your understanding of graph algorithms through a practical application. You will:

- Implement an **undirected weighted graph** data structure using **Adjacency Map Representation**.
- Apply Depth-First Search **(DFS)** to explore the game world.
- Use Breadth-First Search **(BFS)** to find the shortest path to rally allies.
- Utilize **Dijkstra's algorithm** to find the cheapest path to the treasure.

3 Input File Format

3.1 File Structure

The input file consists of two sections: nodes and edges, indicated by `#NODES` and `#EDGES` respectively.

- The `#NODES` section lists all nodes with attributes, as either "start", "ally", or "treasure".
- The `#EDGES` section lists all edges with associated costs.

Each section begins with a comment line and contains entries until the next comment line or the end of the file.

Here's a complete example of the `gameWorld.dat` file:

```
#NODES
Node 1, label=start
Node 2, label=ally
Node 3, label=treasure
Node 4, label=ally
Node 5, label=treasure
Node 6, label=treasure
Node 7, label=treasure
Node 8, label=ally
#EDGES
Edge 1 2, cost=3
Edge 1 4, cost=2
Edge 1 5, cost=4
Edge 1 7, cost=5
Edge 2 4, cost=1
Edge 2 5, cost=1
Edge 2 3, cost=2
Edge 3 5, cost=6
Edge 3 6, cost=11
Edge 4 5, cost=1
Edge 5 6, cost=3
Edge 6 7, cost=2
Edge 6 8, cost=3
Edge 7 8, cost=7
```

4 Tasks

As an adventurer in the world of Graph Explorer, you are tasked with exploring the realm, rallying allies, and uncovering treasures. Each task below represents a crucial part of your journey, requiring strategic thinking and mastery of graph algorithms to succeed.

4.1 Task 1: Explore the Graph

Your first challenge is to explore the unknown territories of the graph. Given a starting node, you must traverse the graph using **Depth-First Search (DFS)**. Document the ids of the nodes in the order they are visited. When faced with multiple unvisited adjacent nodes, **prioritize them in lexicographical order** before pushing them onto the stack. Return the list of nodes visited in order.

Expected Output: The expected output of exploration for the above-given graph example is:

```
['1', '2', '3', '5', '4', '6', '7', '8']
```

4.2 Task 2: Rally Allies

No hero can stand alone. It's time to rally the allies closest to you. Starting from your current location, you need to find the closest allies, **disregarding the weights of the paths.** Use **Breadth-First Search (BFS)** for this task, exploring neighbors layer by layer. When expanding nodes, always **process** them **in lexicographical order**. Output the list of allies sorted first by distance, then by the lexicographical order of their identifiers if distances are equal.

Expected Output: The expected output of rallying allies for the above-given graph example is:

```
['2', '4', '8']
```

4.3 Task 3: Treasure Hunt

Legends tell of treasures scattered across the land, each hidden away and guarded by challenges unknown. Your task is to **find the cheapest way to reach each treasure. Utilizing Dijkstra's algorithm**, calculate the paths to these treasures from your current location, considering the cost of traversing each path as the sum of the costs of the traversed edges. **In the case of ties where multiple paths have the same minimum cost, maintain the first discovered path and avoid updating it upon subsequent tie encounters.** Return a list of tuples, where each tuple contains the path taken to a treasure and the total cost of that path.

Expected Output: The expected output of the treasure hunt task for the above-given graph example is:

```
[(['1', '2', '3'], 5),
 (['1', '4', '5'], 3),
 (['1', '4', '5', '6'], 6),
 (['1', '7'], 5)]
```

5 Class Overview

In this section, the classes that form the backbone of the "Graph Explorer" game are outlined. Each class is designed with specific responsibilities, following the principles of object-oriented programming to create a modular and understandable codebase.

5.1 GameController

The `GameController` class is the central hub of the game, **responsible for controlling the flow of the game, including setup and gameplay**. This class has been **implemented for you**. It uses methods from the `WeightedGraph` and `GraphParser` classes to

navigate and interact with the graph, **orchestrating the game's primary functions: initialization, exploration, rallying allies, and treasure hunting.**

5.2 WeightedGraph

The `WeightedGraph` class represents the game world as a weighted graph. You are to **implement this class**. It utilizes an **adjacency map** for its representation, where each node is stored as follows:

```
# Format: {node_id: {'label': label, 'edges': {neighbor_id: cost}}}
```

Each node in this dictionary includes a label and a dictionary of edges that connect it to other nodes, with the cost associated with each edge. You will need to implement several methods including `dfs`, `bfs`, and `dijkstra` to allow various graph traversal and pathfinding functionalities:

- **add_node (Provided for parsing)**: Adds a node to the graph.
- **add_edge (Provided for parsing)**: Adds an edge between two nodes.
- **dfs**: Performs a depth-first search from a given start node.
- **bfs**: Performs a breadth-first search from a given start node.
- **dijkstra**: Calculates the shortest paths from the start node to all other nodes using Dijkstra's algorithm.

You are required to fully implement the traversal methods, ensuring they function correctly within the provided graph structure. You are free to add as many attributes and methods as you wish.

5.3 GraphParser (Provided)

The `GraphParser` class is provided to you. It is responsible for parsing an input file with `.dat extension` containing the graph's representation and initializing the `WeightedGraph` with nodes and edges based on the file contents. This class simplifies the process of constructing the graph from data, allowing you to focus on implementing graph traversal algorithms.

6 Testing

To assist you in verifying the correctness of your implementations for the `WeightedGraph` methods (`dfs`, `bfs`, and `dijkstra`), a testing notebook, `tester.ipynb`, is provided. This notebook performs unit testing on each of the tasks specified in this assignment:

- **Explore the Graph (DFS)**: Tests the depth-first search functionality.
- **Rally Allies (BFS)**: Tests the breadth-first search to ensure it identifies the shortest paths correctly.
- **Treasure Hunt (Dijkstra)**: Tests the Dijkstra's algorithm implementation for finding the least-cost paths.

The tests compare the outputs generated by your implementation against the expected outputs given in the task descriptions. If your results match the expected ones, the notebook will indicate a provisional pass for that particular test.

Note: The **grade** calculated by this notebook is **provisional** and is based on the operations tested within this notebook. It is unlikely to cover all possible scenarios or test cases. The **final assessment may differ**, and thus, the grade obtained through this notebook should be viewed as indicative and not final. Thus, you are encouraged to **write your own test cases in addition to those provided.**

Note: Be cautious when re-running cells as each execution updates your provisional grade based on the result of each test, which may be misleading.

Note: The overall provisional grade proposed by the tester will be out of 90. The remaining 10 points will be obtained through a `README.MD` file, which should **document your operations, usage instructions, and suggestions for improving the hash functions.** Please refer to the Grading Criteria section for detailed information.

7 README.MD File

As part of your submission, you are required to include a `README.md` file that provides an overview of your implementations and discusses theoretical aspects of your work. This documentation is crucial for evaluating your understanding of the material and your ability to communicate technical information effectively.

Your `README.md` should include the following sections:

- **Classes and Methods Overview:** Provide a detailed description of the classes and methods implemented. Explain the role of each class and method within the application, including how they interact with each other.
- **Discussion of Graph Representations:**
  - **Describe different graph representations:** Discuss at least two types of graph representations.
  - **Compare and Contrast:** Analyze the advantages and disadvantages of each representation in terms of time and space complexity, particularly in relation to operations like adding a vertex, adding an edge, and finding the shortest path.

Note: Use Markdown formatting to enhance the readability of your document. This includes using headers, lists, code blocks, and tables where appropriate.

Grading Criteria

The grading for this assignment totals 100 points, distributed as follows:

- Graph Exploration via DFS: 25 Points
- Rally Allies via BFS: 25 Points

- Treasure Hunt via Dijkstra: 40 Points
- README.MD: 10 Points

Submission

Submissions are through ODTUCLASS. Please package your files into a zip archive with the naming format `<student_id>.zip` including the following files:

- `GameController.py`
- `GraphParser.py`
- `WeightedGraph.py`
- `README.MD`

Ensure that all files are correctly named and contain the necessary implementations as specified in the assignment guidelines. The README file should provide clear and concise documentation of your project. **Ensure your implementation works correctly with the tester.ipynb shared.**

Plagiarism Policy:

For any questions or discussions, if they do not contain specific code snippets etc., please use the discussion thread on `ODTUClass` so that everyone can benefit and be kept up-to-date. In such cases where you need to share specific code/pseudo code (anything not abstract regarding your solution), please do not hesitate to reach out via e-mail. You are encouraged to answer each other's questions and discuss among yourselves provided you do not share any code/pseudocode etc. Such actions of sharing are a direct violation of academic integrity and will be subject to disciplinary action. This also holds for online/AI sources, including Google, ChatGPT, and Copilot. **Make use of them responsibly**- refrain from generating the code you are asked to implement. Remember that we also have access to these tools, making it easier to detect such cases. **In short, please do not resort to practices violating your integrity - we are here to help.**