PA4.1 Graphs

Comprehensive Guide to Implementing Graph Algorithms

Lead TAs: Aamil and Faaiz

November 18, 2024

Contents

1 Overview of the Graph Class Implementation

This manual provides a comprehensive guide to implementing a generic `Graph` class in C++. The implementation is organized into three main tasks, each corresponding to different aspects of the graph's functionality. The class is defined in the `Graph.h` header file, which includes all necessary member functions divided into these tasks.

1.1 Graph.h Header File Structure

The `Graph.h` file defines the `Graph` class template, which manages the vertices and edges of the graph. The class is designed to handle both directed and undirected graphs, as well as weighted and unweighted edges. Below is an overview of how the tasks are organized within the class:

- **Task 1: Basic Graph Operations**
  - **Graph construction, addition and removal of vertices and edges.**
  - **Retrieval functions for vertices and edges.**
  - **Functions to display the graph's adjacency matrix and list.**
- **Task 2: Traversal and Single-Source Shortest Path Algorithms**
  - **Breadth-First Search (BFS) and Depth-First Search (DFS) traversals.**
  - **Dijkstra's and Bellman-Ford algorithms for finding the shortest paths from a source vertex.**
- **Task 3: Advanced Graph Algorithms**
  - **Finding the shortest path between two vertices.**
  - **Generating spanning trees and finding the minimum spanning tree.**
  - **Performing topological sorting.**
  - **Identifying connected and strongly connected components.**

Task 4: Graph Applications

- **Task 4 will be based upon the previous three tasks, PA4.1. This Manual is for PA4.1 (Tasks 1-3). Task 4 will be shared in a few days.**

Each task builds upon the previous one, enhancing the functionality and complexity of the `Graph` class.

2 Task 1: Implementing the Graph Class

In this task, you are required to implement the basic functionalities of the `Graph` class in C++. This includes constructing the graph, adding and removing vertices and edges, retrieving vertices and edges, and displaying the graph's structure through adjacency representations.

2.1 Function 1.1: *Graph()*

- **Description**: Constructs a new graph instance. The graph is not yet defined as weighted or directed yet so assume false for both.
- **Outputs**: Initializes the graph. No return value.

2.2 Function 1.2: *Graph(bool directed, bool weighted)*

- **Description**: Constructs a new graph instance. The graph can be either directed or undirected, and weighted or unweighted based on the parameters provided.
- **Inputs**:
  - *directed* (`bool`):
    - `true`: The graph is directed.
    - `false`: The graph is undirected.
  - *weighted* (`bool`):
    - `true`: The graph supports weighted edges.
    - `false`: The graph is unweighted.
- **Outputs**: Initializes the graph based on the specified parameters. No return value.

2.3 Function 1.3: *addVertex(T data)*

- **Description**: Adds a new vertex to the graph with the specified data.
- **Inputs**:
  - *data* (`T`): The data to be stored in the new vertex.
- **Outputs**: The graph is updated to include the new vertex. No return value.

2.4 Function 1.4: *addEdge(T source, T destination, int weight)*

- **Description**: Adds an edge between two vertices in the graph. If the graph is weighted, the edge will have the specified weight; otherwise, the weight is ignored or set to a default value (e.g., 0).
- **Inputs**:
  - *source* (`T`): The data of the source vertex.
  - *destination* (`T`): The data of the destination vertex.
  - *weight* (`int`): The weight of the edge. Use 0 for unweighted graphs.
- **Outputs**: The graph is updated to include the new edge. No return value.
- **Edge Cases**:
  - If either the `source` or `destination` vertex does not exist in the graph, the function does not add the edge.
  - If an edge between the `source` and `destination` already exists, the function updates the weight if the graph is weighted or ignores the duplicate if unweighted.

2.5 Function 1.5: *removeVertex(T data)*

- **Description**: Removes the vertex with the specified data from the graph, along with all edges connected to it.
- **Inputs**:
  - *data* (`T`): The data of the vertex to be removed.
- **Outputs**: The graph no longer contains the specified vertex and its associated edges. No return value.
- **Edge Cases**:
  - If the vertex with the specified `data` does not exist, the function does nothing.

2.6 Function 1.6: *removeEdge(T source, T destination)*

- **Description**: Removes the edge between the specified source and destination vertices.
- **Inputs**:
  - *source* (`T`): The data of the source vertex.
  - *destination* (`T`): The data of the destination vertex.
- **Outputs**: The specified edge is removed from the graph. No return value.
- **Edge Cases**:
  - If either the `source` or `destination` vertex does not exist, or if the edge between them does not exist, the function does nothing.

2.7 Function 1.7: *getVertex(T data)*

- **Description**: Retrieves the vertex object containing the specified data.
- **Inputs**:
  - *data* (`T`): The data of the vertex to retrieve.
- **Outputs**: Returns a shared pointer to the vertex with the specified data. If the vertex does not exist, it returns a null pointer.
- **Edge Cases**:
  - If no vertex with the specified `data` exists in the graph, the function returns `nullptr`.

2.8 Function 1.8: *getEdge(T source, T destination)*

- **Description**: Retrieves the edge object between the specified source and destination vertices.
- **Inputs**:
  - *source* (`T`): The data of the source vertex.
  - *destination* (`T`): The data of the destination vertex.
- **Outputs**: Returns a shared pointer to the edge between the specified vertices. If the edge does not exist, it returns a null pointer.
- **Edge Cases**:
  - If either the `source` or `destination` vertex does not exist, or if the edge between them does not exist, the function returns `nullptr`.

2.9 Function 1.9: *getAllVertices()*

- **Description**: Retrieves a list of all vertices present in the graph.
- **Inputs**: None.
- **Outputs**: Returns a vector of shared pointers to all vertices in the graph.
- **Edge Cases**:
  - If the graph has no vertices, the function returns an empty vector.

2.10 Function 1.10: *getAllEdges()*

- **Description**: Retrieves a list of all edges present in the graph.
- **Inputs**: None.
- **Outputs**: Returns a vector of shared pointers to all edges in the graph.
- **Edge Cases**:
  - If the graph has no edges, the function returns an empty vector.

2.11 Function 1.11: *getEdges(shared_ptr<Vertex<T>> vertex)*

- **Description**: Retrieves all edges connected to a specific vertex.
- **Inputs**:
  - *vertex* (`shared_ptr<Vertex<T>>`): The vertex whose edges are to be retrieved.
- **Outputs**: Returns a vector of shared pointers to all edges connected to the specified vertex. If the vertex does not exist, it returns an empty vector.
- **Edge Cases**:
  - If the specified `vertex` is `nullptr` or does not exist in the graph, the function returns an empty vector.

2.12 Function 1.12: *getAdjacentVertices(shared_ptr<Vertex<T>> vertex)*

- **Description**: Retrieves all vertices adjacent to the specified vertex.
- **Inputs**:
  - *vertex* (`shared_ptr<Vertex<T>>`): The vertex whose adjacent vertices are to be retrieved.
- **Outputs**: Returns a vector of shared pointers to all adjacent vertices. If the vertex does not exist, it returns an empty vector.
- **Edge Cases**:
  - If the specified `vertex` is `nullptr` or does not exist in the graph, the function returns an empty vector.

2.13 Function 1.13: *getInAdjacentVertices(shared_ptr<Vertex<T>> vertex)*

- **Description**: Retrieves all incoming adjacent vertices to the specified vertex (applicable only for directed graphs).
- **Inputs**:
  - *vertex* (`shared_ptr<Vertex<T>>`): The vertex whose incoming adjacent vertices are to be retrieved.
- **Outputs**: Returns a vector of shared pointers to all incoming adjacent vertices. If the vertex does not exist or the graph is undirected, it returns an empty vector.
- **Edge Cases**:
  - If the specified `vertex` is `nullptr`, does not exist in the graph, or if the graph is undirected, the function returns an empty vector.

2.14 Function 1.14: *getOutAdjacentVertices(`shared_ptr<Vertex<T>>` vertex)*

- **Description**: Retrieves all outgoing adjacent vertices from the specified vertex (applicable only for directed graphs).
- **Inputs**:
  - *vertex* (`shared_ptr<Vertex<T>>`): The vertex whose outgoing adjacent vertices are to be retrieved.
- **Outputs**: Returns a vector of shared pointers to all outgoing adjacent vertices. If the vertex does not exist or the graph is undirected, it returns an empty vector.
- **Edge Cases**:
  - If the specified `vertex` is `nullptr`, does not exist in the graph, or if the graph is undirected, the function returns an empty vector.

2.15 Function 1.15: *isDirected()*

- **Description**: Determines whether the graph is directed.
- **Inputs**: None.
- **Outputs**: Returns `true` if the graph is directed, `false` otherwise.
- **Edge Cases**:
  - If the graph has no vertices or edges, the function accurately reflects the graph's directed or undirected nature based on its initialization.

2.16 Function 1.16: *isWeighted()*

- **Description**: Determines whether the graph is weighted.
- **Inputs**: None.
- **Outputs**: Returns `true` if the graph is weighted, `false` otherwise.
- **Edge Cases**:
  - If the graph has no edges, the function accurately reflects whether it is designed to handle weights based on its initialization.

2.17 Function 1.17: *updateAdjacencyMatrix()*

- **Description**: updates the adjacency matrix for the graph.
- **Inputs**: None.
- **Outputs**: No return value. Internal Adjacency Matrix updated.
- **Edge Cases**:
  - which adjacency matrix to update? (There are two)

2.18 Function 1.18: *getAdjacencyMatrix()*

- **Description**: Prints the adjacency matrix representation of the graph to the console.
- **Inputs**: None.
- **Outputs**: Returns a `vector<vector<int>>` Aka the Adjacency matrix
- **Edge Cases**:
  - If the graph has no vertices, the function indicates that the adjacency matrix is empty.

2.19 Function 1.19: *printAdjacencyMatrix()*

- **Description**: Prints the adjacency matrix representation of the graph to the console.
- **Inputs**: None.
- **Outputs**: Outputs the adjacency matrix to the standard output. No return value.
- **Edge Cases**:
  - If the graph has no vertices, the function indicates that the adjacency matrix is empty.

2.20 Function 1.20: *printAdjacencyList()*

- **Description**: Prints the adjacency list representation of the graph to the console.
- **Inputs**: None.
- **Outputs**: Outputs the adjacency list to the standard output. No return value.
- **Edge Cases**:
  - If the graph has no vertices, the function indicates that the adjacency list is empty.

3 Task 2: Traversal and Single-Source Shortest Path Algorithms

In this task, you are required to implement various traversal and shortest path algorithms within the `Graph` class. These algorithms include Breadth-First Search (BFS), Depth-First Search (DFS), Dijkstra's Shortest Path, and Bellman-Ford Shortest Path. Each function is designed to operate generically with any data type `T` used in the graph's vertices.

3.1 Function 2.1: *BFSTraversal(`shared_ptr<Vertex<T>>` vertex)*

- **Description**: Performs a Breadth-First Search (BFS) traversal starting from the given vertex. BFS explores the graph level by level, visiting all neighbors of a vertex before moving to the next level.
- **Inputs**:
  - *vertex* (`shared_ptr<Vertex<T>>`): The starting vertex for the BFS traversal.
- **Outputs**: Returns a vector of shared pointers to the vertices in the order they were visited during the traversal.
- **Edge Cases**:
  - If the specified `vertex` is `nullptr` or does not exist in the graph, the function returns an empty vector.
  - If the graph has no vertices, the function returns an empty vector.
  - If the starting vertex has no adjacent vertices, the function returns a vector containing only the starting vertex.

3.2 Function 2.2: *DFSTraversal(`shared_ptr<Vertex<T>>` vertex)*

- **Description**: Performs a Depth-First Search (DFS) traversal starting from the given vertex. DFS explores as far as possible along each branch before backtracking.
- **Inputs**:
  - *vertex* (`shared_ptr<Vertex<T>>`): The starting vertex for the DFS traversal.
- **Outputs**: Returns a vector of shared pointers to the vertices in the order they were visited during the traversal.
- **Edge Cases**:
  - If the specified `vertex` is `nullptr` or does not exist in the graph, the function returns an empty vector.
  - If the graph has no vertices, the function returns an empty vector.
  - If the starting vertex has no adjacent vertices, the function returns a vector containing only the starting vertex.

3.3 Function 2.3: *dijkstraShortestPath(`shared_ptr<Vertex<T>>` source)*

- **Description**: Performs Dijkstra's Algorithm starting from the given source vertex. This algorithm computes the shortest paths from the source vertex to all other vertices in a weighted graph with non-negative edge weights.
- **Inputs**:
  - *source* (`shared_ptr<Vertex<T>>`): The source vertex for the algorithm.
- **Outputs**: Returns a vector of shared pointers to the vertices in ascending order of their distance from the source vertex. If the graph contains vertices that are unreachable from the source, those vertices are excluded from the returned vector.
- **Edge Cases**:
  - If the specified `source` is `nullptr` or does not exist in the graph, the function returns an empty vector.
  - If the graph has negative edge weights, Dijkstra's Algorithm may not work correctly, but the function assumes all weights are non-negative.
  - If the graph is empty, the function returns an empty vector.
  - If the source vertex has no outgoing edges, the function returns a vector containing only the source vertex.

3.4 Function 2.4: *bellmanFordShortestPath(`shared_ptr<Vertex<T>>` source)*

- **Description**: Performs the Bellman-Ford Algorithm starting from the given source vertex. This algorithm computes the shortest paths from the source vertex to all other vertices in a weighted graph and can handle graphs with negative edge weights. It also detects negative weight cycles.
- **Inputs**:
  - *source* (`shared_ptr<Vertex<T>>`): The source vertex for the algorithm.
- **Outputs**: Returns a vector of shared pointers to the vertices in ascending order of their distance from the source vertex. If the graph contains a negative weight cycle reachable from the source, the function handles this scenario appropriately based on the implementation (e.g., returns an empty vector or uses exception handling).
- **Edge Cases**:
  - If the specified `source` is `nullptr` or does not exist in the graph, the function returns an empty vector.
  - If the graph contains a negative weight cycle reachable from the source, the algorithm cannot compute shortest paths and handles this scenario accordingly.
  - If the graph is empty, the function returns an empty vector.
  - If the source vertex has no outgoing edges, the function returns a vector containing only the source vertex.

4 Task 3: Advanced Graph Algorithms

In this task, you are required to implement additional graph algorithms within the `Graph` class. These algorithms include finding the shortest path between two vertices, generating spanning trees, performing topological sorting, and identifying connected and strongly connected components. Each function is designed to operate generically with any data type `T` used in the graph's vertices.

4.1 Function 3.1: *shortestPath(`shared_ptr<Vertex<T>>` source, `shared_ptr<Vertex<T>>` destination)*

- **Description**: Finds the shortest path between the source and destination vertices using the appropriate algorithm based on the type of graph (directed/undirected, weighted/unweighted).
- **Inputs**:
  - *source* (`shared_ptr<Vertex<T>>`): The source vertex for the shortest path.
  - *destination* (`shared_ptr<Vertex<T>>`): The destination vertex for the shortest path.
- **Outputs**: Returns a vector of shared pointers to the vertices representing the shortest path from the source to the destination. If no path exists, it returns an empty vector.
- **Edge Cases**:
  - If either the `source` or `destination` vertex is `nullptr` or does not exist in the graph, the function returns an empty vector.
  - If the graph has no vertices or edges, the function returns an empty vector.
  - If the `source` and `destination` vertices are the same, the function returns a vector containing only that vertex.
  - If no path exists between the `source` and `destination`, the function returns an empty vector.

4.2 Function 3.2: *SpanningTrees()*

- **Description**: Finds all the spanning trees of the graph.
- **Inputs**:
  - None.
- **Outputs**: Returns a vector of shared pointers to `Graph` objects, each representing a spanning tree of the original graph. If the graph is disconnected, spanning trees are generated for each connected component.
- **Edge Cases**:
  - If the graph has no vertices, the function returns an empty vector.
  - If the graph is disconnected, the function returns spanning trees for each connected component.
  - If the graph has only one vertex, the function returns a vector containing a single spanning tree with that vertex.

4.3 Function 3.3: *minimumSpanningTree()*

- **Description**: Finds the minimum spanning tree (MST) of the graph using an appropriate algorithm (e.g., Kruskal's or Prim's algorithm).
- **Inputs**:
  - None.
- **Outputs**: Returns a shared pointer to a `Graph` object representing the minimum spanning tree of the original graph. If the graph is disconnected, it returns the minimum spanning trees for each connected component.
- **Edge Cases**:
  - If the graph has no vertices, the function returns `nullptr`.
  - If the graph is disconnected, the function may return multiple MSTs for each connected component or handle it based on implementation.
  - If the graph has only one vertex, the function returns a graph with that single vertex.

4.4 Function 3.4: *topologicalSort()*

- **Description**: Performs a Topological Sort on the graph. This is applicable only to Directed Acyclic Graphs (DAGs) and orders the vertices linearly such that for every directed edge from vertex `u` to vertex `v`, `u` comes before `v` in the ordering.
- **Inputs**:
  - None.
- **Outputs**: Returns a vector of shared pointers to the vertices in topological order. If the graph is not a DAG (i.e., it contains cycles), the function returns an empty vector.
- **Edge Cases**:
  - If the graph has no vertices, the function returns an empty vector.
  - If the graph contains cycles, the function returns an empty vector.
  - If the graph is empty or has no edges, the function returns a vector containing all vertices in any order.

4.5 Function 3.5: *connectedComponents()*

- **Description**: Finds all the connected components of the graph. A connected component is a set of vertices where each vertex is reachable from any other vertex in the same set.
- **Inputs**:
  - None.
- **Outputs**: Returns a vector of vectors, where each inner vector contains shared pointers to vertices that form a connected component.
- **Edge Cases**:
  - If the graph has no vertices, the function returns an empty vector.
  - If the graph is fully connected, the function returns a vector containing a single connected component with all vertices.
  - If the graph has multiple disconnected components, each component is represented as a separate inner vector.

4.6 Function 3.6: *stronglyConnectedComponents()*

- **Description**: Finds all the strongly connected components of the graph. A strongly connected component is a set of vertices in a directed graph where every vertex is reachable from every other vertex in the same set.
- **Inputs**:
  - None.
- **Outputs**: Returns a vector of vectors, where each inner vector contains shared pointers to vertices that form a strongly connected component.
- **Edge Cases**:
  - If the graph has no vertices, the function returns an empty vector.
  - If the graph is a Directed Acyclic Graph (DAG), each vertex forms its own strongly connected component.
  - If the graph has multiple strongly connected components, each is represented as a separate inner vector.

5 Submission Guidelines

- Submit only the required `.cpp` files.
- Zip your submission with the name `PA3_<roll number>.zip`.
- Upload the zip file on LMS before the deadline.
- You have 4 ”free” late days across the semester.
- Penalties for late submissions:
  - 10% penalty for up to 24 hours late.
  - 20% penalty for up to 2 days late.
  - 30% penalty for up to 3 days late.
  - No submissions accepted after 3 days.