INFORMATION
TECHNOLOGY
UNIVERSITY

Software Engineering Department - ITU
SE200L: Data Structures & Algorithms Lab

Course Instructor: Usama Bin Shakeel	**Dated: 13/11/2024**
Teaching Assistant: Zainab Bashir & Ryan Naveed	**Semester: Fall 2024**
Lab Engineer: Sadia Ijaz	**Batch: BSSE2023B**

Lab 11. Prim's Algorithm

Name	**Roll number**	**Report (out of 35)**

Checked on: ______________________________

Signature: __________________________________

1.1 Objective

The objective of this lab is to practice problems related to implementation of Prim's Algorithm.

1.2 Equipment and Component

Component Description	Value	Quantity
Computer	Available in lab	1

1.3 Conduct of Lab

1. Students are required to perform this experiment individually.
2. In case the lab experiment is not understood, the students are advised to seek help from the course instructor, lab engineers, assigned teaching assistants (TA) and lab attendants.

1.4 Theory and Background

Prim's Algorithm is a greedy algorithm used to find the Minimum Spanning Tree (MST) of a weighted, connected graph. An MST is a subset of the edges of the graph that connects all vertices without any cycles and with the minimum possible total edge weight. Prim's algorithm starts with a single node and, at each step, selects the smallest edge that connects a node in the tree to a node outside of it. The algorithm requires efficient data structures to keep track of the minimum edges, such as heaps.

In this lab, you will implement Prim's MST algorithm using C++ by constructing classes to manage nodes, edges, graphs, and minimum heaps. You will build on your previous Graph, Node, and Edge classes, which represent a graph structure. This lab will focus on implementing the MinHeap class to support edge selection in Prim's algorithm and the PrimMST class to execute the algorithm.

Lab Tasks

Task 1:

Implement Graph (Re-use these classes from Lab 9)

Class: Node

- **Data Members**:
  - int data: Value of the node.
  - Node* next: Pointer to the next node in the linked list.
  - Edge* edges: Pointer to the head of the linked list of edges connected to this node.
  - int edgeCount: Number of edges connected to this node.
- **Member Functions**:
  - Node(int val): Constructor that initializes a node with a specified value, setting edges and next to nullptr and edgeCount to zero.

  - int getData(): Returns the node's data value.
  - Node* getNext(): Returns a pointer to the next node.
  - Edge* getEdges(): Returns a pointer to the head of the edges list.
  - int getEdgeCount(): Returns the number of edges.
  - void setNext(Node* nextNode): Sets the next pointer.
  - void setEdges(Edge* edgeList): Assigns a new edge list to this node.
  - void setData(int value): Updates the data value of the node.
  - void incrementEdgeCount(): Increments edgeCount by one.
  - void decrementEdgeCount(): Decrements edgeCount by one.

Class: Edge

- **Data Members**:
  - int destination: The destination node this edge points to.
  - int cost: The cost or weight of this edge.
  - Edge* next: Pointer to the next edge in the list of edges for a node.
- **Member Functions**:
  - Edge(int dest, int c): Constructor that initializes an edge with a destination and cost.
  - int getDestination(): Returns the destination node of the edge.
  - int getCost(): Returns the cost of the edge.
  - Edge* getNext(): Returns the next edge in the list.
  - void setNext(Edge* nextEdge): Sets the next edge pointer.
  - void setDestination(int dest): Sets the destination node for this edge.

Class: Graph

- **Data Members**:
  - Node* nodes: Pointer to the head of the linked list of nodes in the graph.
  - int nodeCount: Total number of nodes in the graph.
- **Member Functions**:
  - Graph(): Initializes an empty graph with nodes set to nullptr and nodeCount to zero.
  - ~Graph(): Destructor to delete all nodes and edges.
  - void addNode(int nodeValue): Adds a new node with the given value, avoiding duplicates.
  - void addEdge(int source, int destination, int cost): Adds an edge from source to destination with a specified cost.
  - int getEdgeCost(int source, int destination): Retrieves the cost of the edge between source and destination.
  - int getEdgeCountForNode(int nodeValue): Returns the edge count for a specified node.
  - int getNodeCount(): Returns the total node count.
  - void updateNode(int oldValue, int newValue): Changes the value of a node.
  - void deleteNode(int nodeValue): Deletes a specified node and its edges.
  - bool hasNode(int nodeValue): Checks if a node exists.
  - void deleteEdge(int source, int destination): Removes an edge from source to destination.

- void display(): Prints each node's value, its edges, and the costs.
- **Node* getNodes():** Returns nodes

Task 2:

Implement the MinHeap Class

Class: **<u>MinHeap</u>**

Purpose: To store edges in a minimum heap structure for efficient retrieval of the smallest edge in Prim's MST algorithm.

- **Data Members**:
  - **Edge** heap:** Array of edge pointers for the heap.
  - **int heapSize:** Number of elements in the heap.
  - **int capacity:** Maximum heap size.
- **Member Functions**:
  - **MinHeap(int capacity):** Constructor to initialize heap with specified capacity.
  - **bool isEmpty():** Returns true if the heap is empty.
  - **void insert(Edge* edge):** Inserts an edge into the heap while maintaining the min-heap property.
  - **Edge* extractMin():** Removes and returns the edge with the smallest cost.
  - **void heapify(int index):** Rearranges elements to maintain the min-heap property starting from a given index.
  - **void swap(int i, int j):** Swaps two elements in the heap array.

Task 3:

Implement Prim's MST Algorithm

Class: **<u>PrimMST</u>**

Purpose: To apply Prim's algorithm on a graph and construct a minimum spanning tree.

- **Data Members**:
  - **Graph* graph:** Pointer to the Graph object.
  - **bool* inMST:** Array to track if a node is included in the MST.
  - **int mstCost:** Total cost of the MST.
- **Member Functions**:
  - **PrimMST(Graph* graph):** Constructor that initializes graph, inMST array, and sets mstCost to zero.
  - **void runPrim(int startNode):** Executes Prim's algorithm starting from startNode, building the MST.

- **void addEdgesToHeap(int node, MinHeap& heap):** Adds all edges connected to node to the heap.
- **int getMSTCost():** Getter for the MST cost
- **bool isNodeInMST(int node):** check if a node is in the MST and return
- **void displayMST():** Displays the MST's edges and total cost.

Instructions:

MinHeap Class Functions:

- **Constructor**: Sets the initial heap size to zero and allocates memory for the heap array.
- **isEmpty**: Returns true if heapSize is zero.
- **insert**: Adds a new edge to the heap while preserving the heap's min-heap structure.
- **extractMin**: Retrieves and removes the edge with the minimum cost.
- **heapify**: Adjusts elements in the heap starting from a specified index to maintain the min-heap property.
- **swap**: Swaps two elements in the heap.

PrimMST Class Functions:

- **Constructor**: Initializes the MST tracking array and sets mstCost to zero.
- **runPrim**: Applies Prim's algorithm, using the MinHeap to track the minimum-cost edges and growing the MST by adding the smallest edge that connects a new node.
- **addEdgesToHeap**: Adds all edges of a specified node to the MinHeap.
- **displayMST**: Outputs the MST edges and displays the total MST cost.

Please read the following instructions carefully:

1. ***Do Not Modify test.cpp:*** *You are strictly prohibited from making any changes to the test.cpp file. This file is designed to test your implementation and any modifications will lead to the assignment being graded as zero.*
2. ***Class Definitions:*** *All class definitions and implementations must be provided solely within the files functions.h and functions.cpp. You are not allowed to create any additional files for your class definitions or implementations.*

Any deviation from these rules, including creating additional files or modifying the test.cpp file, will result in your assignment receiving a grade of zero.

Assessment Rubric for Lab

Performance metric	CLO	Able to complete the task over 80% (4-5)	Able to complete the task 50-80% (2-3)	Able to complete the task below 50% (0-1)	Marks
1. Realization of experiment	1	Executes without errors excellent user prompts, good use of symbols, spacing in output. The testing has been completed.	Executes without errors, user prompts are understandable,minimum use of symbols or spacing in output. Some testing has been completed.	Does not execute due to syntax errors, runtime errors, user prompts are misleading or non- existent. No testing has been completed.	
2. Conducting experiment	1	Able to make changes and answer all questions.	Partially able to make changes and few incorrect answers.	Unable to make changes and answer all questions.	
3. Computer use	2	Document submission timely.	Document submission late.	Document submission not done.	
4. Teamwork	3	Actively engages and cooperates with other group member(s) in an effective manner.	Cooperates with other group member(s) in a reasonable manner but conduct can be improved.	Distracts or discourages other group members from conducting the experiment	
5. Laboratory safety and disciplinary rules	3	Code comments are added and do help the reader to understand the code.	Code comments are added and do not help the reader to understand the code.	Code comments are not added.	
6. Data collection	3	Excellent use of white space, creatively organized work, excellent use of variables and constants, correct identifiers for constants, No line-wrap.	Includes name, and assignment, white space makes the program fairly easy to read. Title, organized work, good use of variables.	Poor use of white space (indentation, blank lines) making code hard to read, disorganized and messy.	
7. Data analysis	4	Solution is efficient, easy to understand, and maintain.	A logical solution that is easy to follow but it is not the most efficient.	A difficult and inefficient solution.	
Total (out of 35):					