CSCE4133/5133 – Algorithms
Fall 2023

Assignment 5
Graph Theory
Minimum Spanning Tree and Shortest Path

Out Date: Nov. 13, 2023
Due Date: **Nov. 27, 2023**

Instructions:

- **Written Format & Template:** Students can use either Google Doc or Latex
- Write your full name, email address and student ID in the report.
- Write the number of "Late Days" the student has used in the report.
- Submission via BlackBoard
- **Policy:** Can be solved individually or in group of two students & review the late-day policy.
- **Submissions:** Your submission should be a PDF for the written section & a zip file with your source code implementation.
- For more information, please visit the course website for the homework policy.

1. Overview

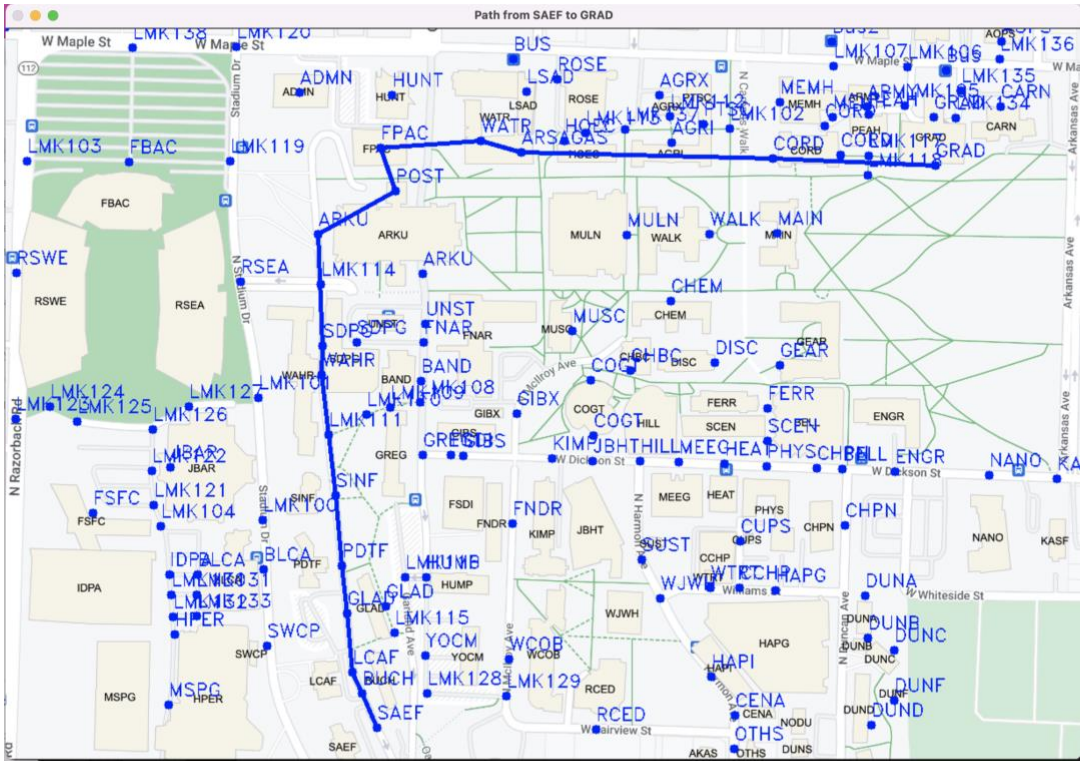

Figure 1: The shortest path from the SAEF building to the GRAD building searched by the Dijkstra's Algorithm

In assignment 4, we studied the ideas of Minimum Spanning and Shortest Path. In this assignment, we will implement the minimum spanning tree algorithms and the shortest path algorithm, i.e., Prim's Algorithm, Kruskal's Algorithm, and Dijkstra Algorithm. In this assignment, you are given a Graph where each node represents a building or a landmark and the weight between two nodes equals the Euclidian distance between them.

In this assignment, you are ***REQUIRED*** to implement ***Prim's Algorithm***, ***Kruskal's Algorithm,*** and ***Dijkstra's Algorithm***. If you think the homework is too long, don't worry, it is long because we provide you with the details of the descriptions. There are some hints that may help to be successful in this homework:

- You should start your homework early.
- You should read the descriptions of homework carefully before you start working on the homework.
- Before implementing the required functions, you should try to compile the source code first. Basically, the source code with an empty implementation can be compiled successfully. This step will help you to understand the procedure of each problem.
- If you forgot these algorithms, you are encouraged to revise the lecture slides.

The source code is organized as follows:

File or Folder	***Required Implementation***	***Purpose***
`assets/`	No	Contain the data of homework
`include/`	No	Contain headers files
`src/`	No	Contain source code files
`bin/`	No	Contain an executable file
`include/graph.hpp`	No	Define the graph structure
`include/sort.hpp`	No	Define the header of sorting algorithms
`src/data_structures`	No	Implement data structures (i.e., linked list, queue, stack, bst, and avl)
`src/algorithms`	No	Implement the algorithms (i.e., Kruskal, Prim, and Dijkstra)
`src/sort`	No	Contain the source files of sorting algorithms
`src/sort/msort.cpp`	No	Implement the Merge Sort algorithm
`src/algorithms/prim.cpp`	Yes	Implement Prim's Algorithm
`src/algorithms/kruskal.cpp`	Yes	Implement Kruskal's Algorithm
`src/algorithms/dijkstra.cpp`	Yes	Implement Dijkstra's Algorithm
`src/main.cpp`	No	Implement the main program
`Makefile.windows`	No	Define compilation rules used on Windows
`Makefile.linux`	No	Define compilation rules used on Linux/Mac OS

2. Implementation

a. Graph and Disjoint Set Usages

Before you start implement algorithms, you first investigate the usages of ***Graph*** and ***Disjoint Set***. This class has been already implemented in the ***src/graph.cpp*** and ***src/data_strutures/disjoint_set.cpp***.

Class	***Method***	***Meaning***	***Sample Code***
Graph	Graph	Initialize a graph with n vertices	Graph G(6);
	insertEdge	Insert an undirected (or directed) edge into the graph	G.insertEdge(0, 1); G.insertEdge(0, 1, directed=true);
	search	Perform the search algorithm to find a path from start to destination	G.search(start, destination, bfs);
	exportEdges	Export a list of edges	std::vector<Edge> edges = G.exportEdges()
DisjointSet	DisjointSet	Create a disjoint set	DisjointSet djs(1000);

	isOnSameSet	Check two vertices is on the same or not	`int value = djs.isOnSameSet(u, v);`
			`if (value == 1)`
			`std::cout << "u and v are on the same set" << std::endl;`

	join	Merge sets of two vertices into the same set	`djs.join(u, v);`

b. Prim's Algorithm

You are given a method in ***src/algorithms/prim.cpp*** as follows:

```
std::vector<Edge> constructMSTPrim(Graph G) {
    std::vector<Edge> edges = G.exportEdges(); // Graph's edges
    // std::priority_queue< Edge, std::vector<Edge>, EdgeKeyComparison > heap;
    // If you want to use heap to optimize the minimum searching, you can use heap defined as above
    //      Insert: heap.push(Edge(u, -1, distance));
    //      Get Minimum: top = heap.top(); u = top.u; distance = top.w;
    //      Remove top: heap.pop(); (this function usually goes after the get minimum method)


    // YOUR CODE HERE


}
```

Given a Graph G, you have to implement Prim's Algorithm and return a list of edges as the minimum spanning tree of G. We provide you a heap implementation as above if you want to optimize your minimum searching algorithm during performing Prim's algorithm.

c. Kruskal's Algorithm

You are given a method in ***src/algorithms/kruskal.cpp*** as follows:

```
std::vector<Edge> constructMSTKruskal(Graph G) {
    std::vector<Edge> edges = G.exportEdges(); // Graph's edges
    // DisjointSet djs(G.n);
    // Use Disjoint Set to check wheter two vertices are on the same set
    // Usage: Check djs.isOnSameSet(u, v); Check is u and v is on the same set or not
    //            djs.join(u, v); Join sets of u and v into the same set


    // YOUR CODE HERE
}
```

Given a Graph G, you have to implement Kruskal's Algorithm and return a list of edges as the minimum spanning tree of G. We provide you a Disjoint Set implementation so that you can use this data structure to check whether two vertices are connected or not.

d. Dijkstra's Algorithm

You are given a method in ***src/algorithms/dijkstra.cpp*** as follows:

```
std::vector<int> searchShortestPath(Graph &G, int start, int destination) {

    // std::priority_queue< Edge, std::vector<Edge>, EdgeKeyComparison > heap;
    // If you want to use heap to optimize the minimum searching, you can use heap defined as above
    //      Insert: heap.push(Edge(u, -1, distance));
    //      Get Minimum: top = heap.top(); u = top.u; distance = top.w;
    //      Remove top: heap.pop(); (this function usually goes after the get minimum method)

    // YOUR CODE HERE
}
```

Given a Graph G, you have to implement Dijkstra's Algorithm and return a list of vertices that is the shortest path from start to destination on graph G. We provide you a heap implementation as above if you want to optimize your minimum searching algorithm during performing Dijkstra's algorithm.

3. Compilation and Testing

In this homework, we use the same environment (Make, GCC/G++, OpenCV Library) used in the previous ones to compile the source code.

In this homework, ***you use only a single Makefile***. It will automatically detect your Operator System (e.g., Linux, MacOS, or Windows) and configurate the compilation rules. For Windows users, you have to make sure that you set the correct paths for MINGW_BIN and OPENCV_DIR:
MINGW_BIN=<path to mingw64>/bin
OPENCV_DIR=<path to opencv>/build/install
For example, if we stored mingw64 and opencv in C:/, we edit these paths as follows:
MINGW_BIN=C:/mingw64/bin
OPENCV_DIR=C:/opencv/build/install

OpenCV Library: In our homework, OpenCV is OPTIONAL and only used for the visualization purpose, you can work on your homework WITHOUT installing and using OpenCV. If you do not want to use OpenCV for visualization, open file ***Makefile*** and edit line ***OPENCV=1*** to ***OPENCV=0.***

To compile and test your source code, you open the terminal and go to the source code folder and type the following commands:

- **make prim (**or **mingw32-make prim** on Windows): To compile and run Prim's algorithm.
- **make kruskal (**or **mingw32-make kruskal** on Windows): To compile and run Kruskal's algorithm.
- **make dijkstra (**or **mingw32-make dijkstra** on Windows): To compile and run Dijkstra's algorithm.

If you compile and run successfully, you should pass the unit test of the sorting algorithm and are going to see an output as follows:

Prim	Perform unit test of the Prim's algorithm Total Cost of Minimum Spanning Tree: 16 Total Cost of Minimum Spanning Tree: 9315
Kruskal	Perform unit test of the Kruskal's algorithm Total Cost of Minimum Spanning Tree: 16 Total Cost of Minimum Spanning Tree: 9315
Dijkstra	Perform unit test of the Dijkstra algorithm Path from 0 to 5: 0 1 2 4 5 Path from SAEF to detination: SAEF -> BUCH -> LCAF -> GLAD -> PDTF -> SINF -> LMK111 -> WAHR -> SDPG -> ARKU -> POST -> FPAC -> WATR -> ARSAGAS -> CORD -> GRAD

For Windows users, if you would like to compile with OpenCV, you have to copy the following files from ***C:\opencv-3.4.13\build\bin*** to your Homework folder:

- libopencv_core3413.dll
- libopencv_highgui3413.dll
- libopencv_imgcodecs3413.dll
- libopencv_imgproc3413.dll