**Project 4: Shortest Paths in Weighted Graphs**

***Weighted graph classes and shortest path algorithms***

**Version 06/23/19**

**Educational Objectives:** After completing this assignment, the student should be able to accomplish the following:

1. Give detailed descriptions of weighted graph representations
2. Using weighted graph representations, give details of the following algrithms, including all components of each algorithm (assumptions, outcomes, body, proof of correctness, runtime, runspace, and proofs):
   1. Kruskal's Minimum Spanning Tree algorithm
   2. Prim's Minimum Spanning Tree algorithm
   3. Dijkstra's Single-Source-Shortest-Paths [SSSP] algorithm
   4. Belman-Ford SSSP algorithm
3. State and prove the Relaxation Theorem

**Operational Objectives:**

1. Develop a graph framework to accommodate weighted edges in both undirected and directed graphs.
2. Use the framework to implement the following algorithms:
   1. Kruskal's Minimum Spanning Tree algorithm
   2. Prim's Minimum Spanning Tree algorithm
   3. Dijkstra's Single-Source-Shortest-Paths [SSSP] algorithm
   4. Belman-Ford SSSP algorithm

**Deliverables:**

**wgraph.h # weighted graph framework**

**kruskal.h # contains algorithm Kruskal<G>**

**prim.h # contains algorithm Prim<G>**

**dijkstra.h # contains algorithm Dijkstra<G>**

**bellford.h # contains algorithm BellFord<G>**

**log.txt # work log / test diary**

**General Discussion**

Note the restriction on use of std:: libraries (under Requirements).

This is a fairly large development project that is somewhat open-ended. The bulk of the code required is in design and implementation of the weighted graph classes. The four algorithms operate on objects from that framework. Therefore it is extremely important that you test the graph classes thoroughly to eliminate the possibility that errors in the outcomes of the algorithms are due to faulty graph classes.

To get started, you need to develop a weighted graph framework for weighted graphs, along the lines of the unweighted version in LIB/graph/graph.h. You need to represent weights of edges. There are at least three ways to do this:

1. Maintain a mapping **weight\_:{edges} -> {real numbers}**.
2. Define an edge class that has 2 vertices and a weight as members. (Use or derive from fsu::Edge<N>.)

These each have advantages and dangers.

**Option 1:** Weight as a mapping **weight\_:{edges} -> {real numbers}**

A strong advantage for following option #1 is that the weighted graph classes can re-use code from the unweighted classes, as follows:

**namespace fsu**

**{**

**template < typename N > ALUG**

**class ALUWGraph : public ALUGraph <N> / \**

**{ ALDG ALUWG**

**... \**

**}; ALDWG**

**template < typename N > blue = defined in graph.h**

**class ALDWGraph : public ALDGraph <N> red = defined in wgraph.h**

**{**

**... (Option 1)**

**};**

**} // namespace fsu**

See the [weighted graph notes](http://www.cs.fsu.edu/~lacher/courses/COP4531/notes/graphs2.html) for more detail.

**Option 2:** Weight as a component of an Edge class

In option 1, edges are represented logically using adjacency lists of vertices, but edges have no concrete representation as objects. In option 2 edges are actual objects consisting of two vertices and a weight. The option 1 adjacency lists of vertices, as in the base class unweighted cases, are replaced with lists of pointers or references into an inventory of actual edge objects. Thus it is not practical to derive weighted graph classes from the unweighted versions as we can in option 1. We can still take advantage of code re-use by deriving the directed case from the undirected:

**namespace fsu**

**{**

**template < typename N > ALUG ALUWG**

**class ALUWGraph / /**

**{ ALDG ALDWG**

**...**

**};**

**template < typename N > blue = defined in graph.h**

**class ALDWGraph : public ALUWGraph <N> red = defined in wgraph.h**

**{**

**... (Option 2)**

**};**

**} // namespace fsu**

Moreover, the API and all of the code implementing code, while not re-usable, is easily adapted to the new model.

In #2 you will need an "edge inventory" (a collection of Edge objects) and a way to refer into that inventory when defining a graph, so that there is never more than one copy of a given edge object anywhere. (E.g., make an adjacency list of pointers into inventory.) This is easier to manage for directed graphs. For undirected graphs you have to be careful that adjacent vertices refer to the same edge (not 2 copies), and ensure that {x,y} == {y,x} with un-ambiguous weight. Finally, ensure that the edge inventory has no redundancies.

**For this project, follow Option 1.**

**Phase 1: The weighted graph and digraph classes**

The project cannot move to graph search until the supporting weighted graph framework has been built, tested, and certified correct and bug-free. Start on this phase ASAP with a class and test design.

Use the names **ALUWGraph** and **ALDWGraph** for the undirected and directed classes.

One weighted graph utility you will need is a function

**template < class G >**

**bool WLoad (std::istream& inStream, G& g);**

that instantiates a weighted graph from data in a file (after first skipping file documentation, lines that begin with '#'). This can be modelled on the Load function for unweighted graphs.

WLoad will of course need a file specification to work with:

**#**

**# documentation begins with '#' at the first character**

**# and continues as long as lines begin with '#'**

**#**

**# first entry is unsigned int n = number of vertices**

**# followed by triples x y w**

**# x,y represent vertices of edge from x to y (unsigned < n)**

**# w is a real number representing weight of edge**

**#**

**n**

**x y w**

**x y w**

**x y w**

**...**

**x y w**

As in the unweighted case, the same file can represent either a directed or an undirected graph, determined by the type being loaded into.

**Phase 2: Kruskal and Prim**

These algorithms should follow the general model we use for BreadthFirstSurvey: a template class that takes a graph reference in its constructor. Follow the models discussed in detail in the [Weighted Graphs Notes](http://www.cs.fsu.edu/~lacher/courses/COP4531/notes/graphs2.html).

**template < class G>**

**class Kruskal**

**{**

**typedef G Graph**

**typedef typename G::Vertex Vertex;**

**typedef fsu::Edge<Vertex> Edge;**

**typedef fsu::Vector<Edge> Container;**

**typedef fsu::GreaterThan<Edge> Predicate;**

**typedef fsu::PriorityQueue<Edge,Container,Predicate> PQ;**

**public:**

**// constructor initializes all class variables in init list**

**Kruskal (const G& g);**

**// implementing the algorithm**

**void Init(bool verbose = 0); // preps class variables for execution of algorithm**

**void Exec(bool verbose = 0); // runs algorithm**

**// extracting information**

**const fsu::List<Edge>& MST() const;**

**double Weight() const;**

**... // expand API optional**

**private:**

**const Graph& g\_;**

**...**

**PQ pq\_;**

**};**

**template < class G>**

**class Prim; // same API as Kruskal**

The goal is for the following kind of declaration to work:

**typedef size\_t Vertex; // template argument for N**

**typedef fsu::ALUWGraph <Vertex> WGraphType; // ALUWGraph**

**WGraphType g;**

**fsu::Prim <WGraphType> prim(g);**

The above should create an undirected adjacency list weighted graph g and a Prim<G> algorithm object prim that operates on g.

The method Exec() should play the equivalent role in all of the algorithms. For example, after the declarations above,

**prim.Init();**

**prim.Exec();**

should be the calls that invoke the Prim process.

**Note:** You may implement the "lazy" versions of Prim and Dijkstra (i.e., using a min-heap priority queue without the extra functionality of "decrease\_key").

**Phase 3: Dijkstra and Bellman-Ford**

**template < class G>**

**class Dijkstra**

**{**

**typedef G Graph**

**typedef typename G::Vertex Vertex;**

**... // more definitions optional**

**public:**

**// constructor initializes all class variables in init list**

**Dijkstra (const G& g);**

**// implementing the algorithm**

**void Init(Vertex source); // preps variables for startup of algorithm**

**void Exec(); // executes algorithm**

**// extracting information**

**const fsu::Vector<double>& Distance() const;**

**const fsu::Vector<size\_t>& Parent() const;**

**void Path(Vertex x, fsu::List<Vertex>& path) const;**

**... // expand API optional**

**private: // data**

**const Graph& g\_;**

**...**

**private: // methods**

**void Relax(Vertex v)**

**...**

**};**

**template < class G>**

**class BellFord; // same API as Dijkstra**

**Procedural Requirements**

1. The official development/testing/assessment environment is specified in the Course Organizer. Code should compile without warnings or errors.
2. In order not to confuse the submit system, create and work within a separate subdirectory cop4531/proj4.
3. Maintain your work log in the text file log.txt as documentation of effort, testing results, and development history. This file may also be used to report on any relevant issues encountered during project development.
4. **General requirement on genericity.** Use generics whenever possible. For example:
   1. If you need to sort, use a generic sort algorithm or a container with built-in sorting.
   2. If a sort requires a specialized notion of order, create a function object that captures the desired order property.
   3. If you need a graph algorithm, use a component of the fsu graph library.
   4. In general, whenever a generic algorithm exists that can be deployed, do not circumvent that with specialized one-off code.
   5. Carefully choose all containers as the most appropriate for a particular purpose.

In short: re-use components from LIB (or your own versions) whenever possible. Don't re-invent the wheels.

1. Begin by copying all files in LIB/proj4 into your project directory. You should see at least these:
2. **deliverables.sh # submission configuration file**
3. **fwgraph.cpp # general test harness for option 1 weighted graph classes**
4. **ranwgraph.cpp # random weighted graph generator**
5. **ranwgraph\_ER.cpp # random weighted Erdos-Renyi graph generator**
6. **edge.api # copy of LIB/graph/edge.h - defines Edge class template**
7. **mst.cpp # mst client program**
8. **sssp.cpp # sssp client program**
9. **makefile # builds essentials**
10. Follow these steps when you are ready to submit:
    1. Be sure that you have established the submit script LIB/scripts/submit.sh as a command in your ~/.bin directory. [We are currently using version 2.0.]
    2. Copy the submit configuration script LIB/proj4/deliverables.sh into your project directory. [This is an important step, as deliverables may have changed.]
    3. Submit the project using the command submit.sh while in your project directory and logged in to shell or quake. Pay attention to the screen, which will warn you about missing deliverables and incorrect login.
    4. Check your CS email for feedback. Revise and re-submit the project if appropriate.

***Warning:****Submit scripts do not work on the program and linprog servers. Use shell.cs.fsu.edu or quake.cs.fsu.edu to submit projects. If you do not receive the second confirmation with the contents of your project, there has been a malfunction.*

**Code Requirements and Specifications**

1. These libraries may NOT be used:
2. **<string>**
3. **<set>**
4. **<unordered\_set>**
5. **<map>**
6. **<unordered\_map>**
7. **<algorithm>**

There are equivalent components of the fsu::library in ~cop4531p/LIB that may be used.

1. **Verbose Behavior.** For both Kruskal and Prim, the following should be the last few lines of code in Init and Exec:
2. **Init(bool verbose)**
3. **{**
4. **...**
5. **if (verbose)**
6. **{**
7. **std::cout << "Init: Initial edges in PQ: " << pq\_.Size() << '\n';**
8. **if (pq\_.Size() < 20)**
9. **{**
10. **std::cout << "PQ.Dump():\n";**
11. **pq\_.Dump(std::cout, '\n');**
12. **std::cout << '\n';**
13. **}**
14. **}**
15. **}**
16. **Exec(bool verbose)**
17. **{**
18. **...**
19. **if (verbose)**
20. **{**
21. **std::cout << "Exec: Remaining edges in PQ: " << pq\_.Size() << '\n';**
22. **if (pq\_.Size() < 20)**
23. **{**
24. **std::cout << "PQ.PopOut():";**
25. **pq\_.PopOut(std::cout, '\n');**
26. **std::cout << '\n';**
27. **}**
28. **}**
29. **}**

Here pq\_ is the private class variable storing the priority queue used in algorithm control.

1. Kruskal, Prim, Dijkstra, and BellFord should faithfully implement the algorithm of the same name, as discusses in the Graph notes and textbook. Specifically: output should match exactly that of the benchmark programs in LIB/area51.