CSE 101 Winter 2018
Programming Assignment (PA) Two

Due: Friday February 9, 11:59 PM PST
Link to starter code: https://github.com/UCSD-CSE101/W18-PA2

PLEASE READ THE FAQ BEFORE POSTING ON PIAZZA
https://docs.google.com/document/d/1zjgCRHbtzaTpb9gs3XTt50tFW5xXEh_X1_jfdN754ao/edit?usp=sharing

Changes to Graph.hpp
In PA2, you will be dealing with graphs with weighted edges. We have added some new things to **Graph.hpp** to help you out with your implementations.

Edge: This class represents an edge from a source (src) and a destination (dest). The weight of this edge is stored within the **Graph** class as a map from edges to weights.

Weights: This map is a new field in the **Graph** class, and maps from edges to weights. These weights can be accessed and changed using **get_weight()** and **set_weight()**. Note that graphs given to you will already have their edge weights initialized.

Distance: Distance is a new field added to the Vertex class. You can use this field as you see fit, but it will be most useful in keeping track of some running distance value. It is initialized to **FLT_MAX**, the maximum float value possible. Distances can be reset to this **FLT_MAX** value using **clear_distance()**

Prev: Although it is not necessary to construct the shortest path or minimum spanning tree, it may or may not be useful for you to keep track of which edges have been added by setting this field. We will initialize this to the self-same vertex id at the start but you can change it when you find which vertex should be connected in your tree.

Exchange: This is an object used to represent the exchange rates between different cryptocurrencies in question 4. For example, if I have an input currency (in) of Bitcoin, and an output currency (out) of Ethereum with a rate (rate) of 10.0, I could exchange 1 Bitcoin for 10.0 Ethereum (before any fees are applied.)

Fees: Fees for each cryptocurrency are not represented as an explicit object in **Graph.hpp**, but are passed into the function in question 4 as a map from strings (the name of the currency) to a float (the fee amount).

Alarm: This is a utility class that represents setting an "alarm" as described in the explanation for Dijkstra's algorithm. The Alarm class has a src and a dest similar to **Edge**, with the addition of a **time** field which represents **when** we might want to explore this edge and its vertices. We have also overridden the < operator which can be used to create a min-priority queue for sorting these Alarms by their times. Although the concept of alarms was used to explain Dijkstra's algorithm, the general concept of sorting edges to explore may be useful in Prim and PrimDijk as well.

A priority queue in C++ requires **template parameters**, with an optional third argument. These arguments are: the type of stored elements, a container to store the elements, and an optional comparator that instructs the priority queue on how to order elements. Since we have overriden the comparison operator in Alarm for you, you can create an Alarm **min-priority queue** with:

```
std::priority_queue<Alarm<T>, std::vector<Alarm<T>>> pq;
```

This line of code will create a priority queue that contains type **Alarm<T>**, uses a **vector<Alarm<T>>** for storage, and uses the overridden default operator< found in the Alarm class for ordering.

Take the time to familiarize yourself with the structure and functions in **Graph.hpp** before beginning any of the questions.

graph_gen.py

To help you create graphs to test with in question 3, we have provided a python script to generate graphs. Running python graph_gen.py --help will give you instructions on how to use it:

```
usage: graph_gen.py [-h] -n NUM -p PROB -w WEIGHT

optional arguments:
  -h, --help  show this help message and exit
  -n NUM      the number of vertices in the graph
  -p PROB     probability that an edge exists between vertex u and v
              -- must be in range (0.0, 1.0]
  -w WEIGHT   Max edge weight. Weights will be randomly selected from
              the range [1.0, w].
```

Generated graphs will be in the graphs folder. These will mainly be used in Question 3, but you can also use them to test Dijkstra's and Prim's in Questions 1 and 2 (Note that it will not tell you what the correct cost of an MST or SPT will be, only generate a graph file.)

Question 1: Dijkstra's Algorithm Implementation [20 Points]

Input: Undirected connected graph ***G*** with weighted edges, and a starting vertex ***u***
Output: Cost of the shortest-path tree rooted at ***u***

Given an undirected connected graph ***G*** with weighted edges, along with an identified source vertex ***u***, your task is to **calculate the total cost of a shortest-paths tree (SPT) rooted at *u***. An SPT is a spanning tree of **G** rooted at **u** such that the path from **u** to any other vertex **v** is the shortest path cost from **u** to **v** in **G**. The cost of a path is the sum of all edge weights along the path. The total cost of an SPT is the sum of all edge weights in the tree (NOT the sum of costs of all paths from root to other vertices). Use **Dijkstra's algorithm** to calculate the cost of the SPT.

You are guaranteed that ***G*** contains no negative-weight edges, and that ***G*** is a connected graph. Since edges are undirected, the vertex IDs of two adjacent vertices are available in each other's edges set. You are also guaranteed that ***G*** contains no self-loops.

For this question, the template type for Graph is <int> and it must be able to operate on very large graphs, so no recursion! Note that we are only looking for the **scalar cost** of the SPT. (It is not necessary to strictly build the SPT in order to calculate its cost.) The tester will read the input file into an undirected graph that will be passed to your algorithm.

To Make:

```
make TestDijkstra
```

To Run:

```
build/TestDijkstra testcases/input_file
```

Example Input:

```
1          The first line is the ID of the root vertex
1 2 5.0    for the SPT. Each subsequent line represents
1 3 5.0    an undirected edge and an edge weight.
1 4 5.1    (<-- This is an edge between 1 and 4 with weight 5.1)
2 4 2.2
```

Example Output:

```
The shortest-paths tree cost is: 15.1
```

Question 2: Prim's Algorithm Implementation [20 Points]

Input: Undirected connected graph ***G*** with weighted edges, starting vertex ***u***
Output: Cost of minimum spanning tree rooted at ***u***

Given an undirected graph ***G*** with weighted edges, calculate the total cost of the minimum spanning tree (MST) rooted at ***u***. An MST is spanning tree of **G** rooted at **u** with the minimum possible total edge weight. The total cost of an MST is the sum of all edge weights in the tree. Note that the SPT and MST for any graph **G** may be different and have different total costs! Use **Prim's Algorithm** (which is similar to Dijkstra's) to calculate the cost of the MST.

Just like in Question 1, edge weights are represented with float values. You are guaranteed that ***G*** contains no negative edges, is connected, and has no self-loops.

For this question, the template type for Graph is <int> and it must be able to operate on very large graphs, so no recursion! All given constraints on ***G*** are the same as in Question 1. Please also remember that you do not have to strictly build the MST rooted at ***u*** to find its scalar cost.

To Make:

```
make TestPrim
```

To Run:

```
build/TestPrim testcases/input_file
```

Example Input:

```
1          The first line is the ID of the root vertex
1 2 5.0    for the SPT. Each subsequent line represents
1 3 5.0    an undirected edge and an edge weight.
1 4 5.1    (<-- This is an edge between 1 and 4 with weight 5.1)
2 4 2.2
```

Example Output:

```
The minimum spanning tree cost is: 12.2
```

Question 3: Analysis of Prim's MST vs. Dijkstra's SPT [30 Points]

Through Question 1 and 2, you've become familiar with the algorithms for Dijkstra's SPT and Prim's MST. Notice that there is tradeoff between the "lightness" of Prim's MST and the "shallowness" of Dijkstra's SPT.

Please review the Prim-Dijkstra slide from Lecture 6: Slide 29 in http://vlsicad.ucsd.edu/courses/cse101-w18/slides-w18/cse101-w18-Lecture6-bfs-sp-dijkstra.pdf

For this question, you will implement a hybrid of the Dijkstra and Prim algorithms, where a conditional weight ***c*** represents the degree to which existing source-to-vertex pathlength in the growing tree ***T*** affects the choice of the next edge. ***c*** is a ratio between 0.0 and 1.0.

When ***c*** is 0.0, the previous path cost leading up to an edge is not considered at all, so only the current edge weight is used. The resulting tree matches that of Prim's MST algorithm.

When ***c*** is 1.0, the previous path cost leading up to an edge is fully considered, so the entire path weight is used. The resulting tree matches that of Dijkstra's SPT algorithm.

In this last question, you will analyze how changing **c** changes the cost of the tree generated on graphs of different densities. Put another way, you will experimentally study the behavior of the Dijkstra-Prim hybrid as the ***c*** value is changed, and as characteristics of the input graph are changed. The deliverable is both your code in PrimDijk.cpp and a **write-up PDF** (max two pages) with your full name(s) & PID(s). As before, the code will be turned in using "make turnin", but the PDF will be turned in on **Gradescope**.

Use the graph_gen.py script (described at the beginning of the write-up) to create graphs of different densities. The **density** of a graph is how many edges exist in the graph relative to the maximum number possible. We can adjust the density by passing in different values for the **p** parameter in graph_gen.py (which represents the probability of an edge existing between any two vertices). A sparser graph has a lower **p**-value (closer to 0.0), and a denser graph has a higher **p**-value (closer to 1.0).

Using ***n* = 2000** and ***w* = 10**, compare your output using the different combinations of **p**-values [0.1, 0.3, 0.5, 0.7, 0.9], and **c**-values [0.0, 0.2, 0.4, 0.6, 0.8, 1 .0]. You **must** include an appropriate data table with the results of your program when executed with the above parameters for the values of ***n***, ***w***, ***p***, and ***c*** (note: we should see 30 data cells in your table). You should also include a detailed analysis of your results, explaining why you see the values you do in your table as you change the value of **c** and the value of **p**, and what this means in relation to Prim's and Dijkstra's algorithm.

To Make:

```
make TestPrimDijk
```

To Run:

```
build/TestPrimDijk testcases/input_file c
```

(***c*** *is a floating point value between 0.0 and 1.0)*

Example Input:

```
0                // excerpt from a generated file
2 1 4.0482       // all trees are rooted at node 0
3 0 5.0054
4 0 5.6284
5 1 1.2790
5 3 4.0352
7 3 7.8317
7 5 5.4047
...
```

Example Output

```
build/TestPrimDijk graphs/0.1_1000_10.0 0.2
The cost of Prim-Dijk with c=0.2 is: 1184.4
```

Your exact cost may vary from the example above due to differences in the generated graph from graph_gen.py. You can also use the graph generator to test your other implementations for Questions 1 and 2 to make sure that they still behave as expected on larger graphs. You may run into quota issues on ieng6 when working with the denser graphs, so be sure to run `make clean_graphs` to clear the generated graphs folder if you are done testing the graphs in there.

Since the code is hard to verify on such large graphs, we have provided some example graphs in the repository. You can use these example graphs to sanity check with the values below using a **c**-value of 0.25. Your output may vary by 1 or 2 but a large difference in the 10's or 100's might mean you should take another look at your code.

n = 1000, w = 10

p	0.1	0.3	0.5	0.7	0.9
Prim (c=0)	1104.62	1033.43	1020.02	1014.8	1010.84
PrimDijk (c=0.25)	1205.93	1119.13	1097.1	1093.48	1090.3
Dijkstra (c=1.0)	1549.1	1395.71	1337.97	1292.92	1279.39

Question 4: Cryptocurrency Arbitrage [30 Points]

Input: List of exchanges ***E***
Map from cryptocurrency to the fee associated with it
Output: Boolean for whether a profit can be made in the current circumstances

With the rise in popularity and trading volume of cryptocurrencies, you decide to join the largest crypto exchange on the web, **Bitvest**. Since these currencies are decentralized and not backed by any government or bank, their prices and exchange rates fluctuate rapidly based on supply and demand. You, a savvy investor, would like to determine, at any given time, if there is a series of trades between cryptocurrencies you can execute to end up with more currency than you begin with. However, since you want to make your trades fast, you have to pay a fraction of your trade amount as a fee to ensure your transactions are verified on both blockchains.

Here's an example. I have 10 BTC and want to trade for ETH, which has an exchange rate of 10, or one BTC to 10 ETH. However, making any transaction with BTC incurs a 0.004 (or 0.4%) fee, and any transaction with ETH incurs a 0.001 (or 0.1%) fee. I first pay 0.004 of my BTC to verify on the BTC blockchain, leaving me with 10*(1-0.004) = 10*(0.996) = 9.96 BTC. I then trade my remaining BTC for ETH at an exchange rate of 10, meaning I get 9.96*10=99.6 ETH. Finally, to verify my ETH transaction on the ETH blockchain, I pay 0.001 of my ETH, leaving me with 99.6*(1-0.001) = 99.6*(0.999) = 99.5 ETH. This calculation can be simplified as follows: 10 * (1-0.004) * 10 * (1-0.001) = 99.5

I continue to make trades in this manner until I end up with more BTC than I end up with (hopefully).

Note that the verification fees are paid for every trade involving a particular currency. This means if I now make a trade from ETH to BCH, I must again pay 0.001 of my ETH before exchanging, then afterwards pay the BCH fee.

Your ultimate goal is to navigate the markets and determine if there are any imbalances in trades and fees such that you can trade and end up with more money than you start with. Note that you do not have to identify the series of trades necessary, just whether or not such a trade is possible.

To Make:

```
make TestBitvest
```

To Run:

```
build/TestBitvest testcases/input_file
```

Example Input (profitable):

```
BTC 0.004       <-- Trading to or from BTC incurs a 0.004 fee.
ETH 0.001       This will be represented in the fees map by a
BCH 0.002       decimal value rather than a percentage

BTC ETH 10.0    <-- BTC->ETH exchange rate is 10.0
ETH BTC 0.01
ETH BCH 5.0
BCH BTC 0.022   // BTC->ETH->BCH->BTC yields a profit!
```

Example Output (unprofitable):

```
Trade status: 1  // Profit!
```

Let's examine this exchange more closely to determine what makes it profitable.

Suppose we start with 100 BTC.
BTC->ETH: 100 * (1-0.004) * 10.0 * (1 - 0.001) = 995.04 ETH
ETH->BCH: 995.04 * (1-0.001) * 5.0 * (1 - 0.002) = 4960.28 BCH
BCH->BTC: 4960.28 * (1-0.002) * 0.022 * (1 - 0.004) = 108.47 BTC

Thus starting from 100 BTC, we have made trades and ended up with 8 more BTC than we started with!

Example Input (unprofitable):

```
BTC 0.004
ETH 0.001
BCH 0.002

BTC ETH 10.0
ETH BTC 0.01
```

```
ETH BCH 5.0
BCH BTC 0.02   // No series of trades makes a profit
```

Example Output (profitable):

```
Trade status: 0  // No profit!
```

Let's take a look again at the same series of trades. Note that BCH is now worth less BTC since it has a lower exchange rate.

Suppose we start with 100 BTC.
BTC->ETH: 100 * (1-0.004) * 10.0 * (1 - 0.001) = 995.04 ETH
ETH->BCH: 995.04 * (1-0.001) * 5.0 * (1 - 0.002) = 4960.28 BCH
BCH->BTC: 4960.28 * (1-0.002) * 0.02 * (1 - 0.004) = 98.61 BTC

After making these trades we have lost 1.4 BTC! The market in this state cannot be used to make a profit through arbitrage.