Algorithms
Homework 10

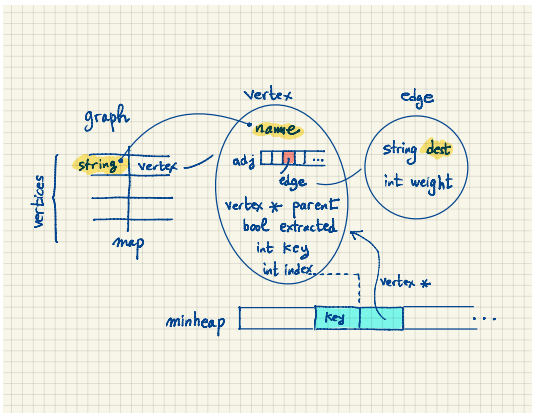
Saad Mneimneh
Computer Science
Hunter College of CUNY

Problem 1: Prim's and Dijkstra's algorithms (Optional coding exercise in C++)

The purpose of this problem is to create a basic class structure for a graph and implement Prim's and Dijkstra's algorithms (they are almost the same). Although this makes the problem simply a copy of some algorithm into code without further intervention or thought, it should give a good idea about how to handle graphs when it comes to C++ code and its STL. Keep in mind that the approach outlined here is not unique.

A minheap implementation has been provided for you in `minheap.h` and `minheap.cpp`. The entries in the heap are pairs. For the purpose of this problem, we assume that the first part of the pair is an integer (the key), and the second part is a pointer to a vertex. An appropriate instantiation of the template has been made in `minheap.cpp`. The vertex object in turn will have an integer index for its position in the heap. The heap has been implemented to update that index, so you do not need to maintain it yourself.

Here's the "architecture" behind the class structure for graph. The code is given in `graph.h`, `graph.cpp`, `minheap.h`, `minheap.cpp`, and `graph_main.cpp` (with a simple `main` function) (the file `graph_main2.cpp` is an alternative implementation that creates the two graphs we have seen in lectures).



Each vertex is identified by its name, which is a string object. The graph maintains a hash table of all the strings. An entry in the hash table is (as usual) a pair. The first part of the pair is, therefore, the string. The second part of the pair is the vertex object. Each vertex has a vector of edges, and an edge defines a destination vertex (by its name) and a weight (an integer).

For instance, to obtain a vertex for a given name, one could do the following:

```
vertex& v = g.vertices[name];
```

or if `x` is a pair in `vertices`

```
vertex& v = x.second;
```

and use `v.adj` to obtain the list of edges for that vertex.

Furthermore, each vertex has three additional attributes: `key`, which is an integer, `parent`, which is a pointer to a vertex, and `extracted`, which is a bool. These are useful for Prim's and Dijkstra's algorithms. Finally, as mentioned earlier, the attribute `index` give the vertex position in the min-heap (for the `decrease_key` operation).

(a) Familiarize yourself with the code, then compile the program and run it. The functions to generate and output the graph have been implemented for convenience. The vertices will have names `a`, ... , `z`, `aa`, ..., `az`, `ba`, ..., `bz`, ...

(b) Add two member functions in `graph`.

```
void add_vertex(const string& s);
void add_edge(const string& s, const string& t, int w);
```

The two functions don't need to make any kind of checks; for instance, whether a vertex or an edge already exists. However, to add an edge, you must check whether the graph is undirected or directed. If undirected, you have to add the edge in both directions. The code in `graph_main2.cpp`

creates two specific graphs using these functions.

(c) Implement the `graph` member function MST (Minimum Spanning Tree) using Prim's algorithm.

(d) Implement the `graph` member function SSSP (Single Source Shortest Path) using Dijkstra's algorithm.

In both cases, think about how you should initialize the vertex keys to "infinity". When done with the implementation, you can use `graph_main2.cpp` to check whether your MST and SSSP work correctly. You can also generate a random small graph in `graph_main.cpp`, say with 5 to 10 vertices, and test your code (to output the result, you can use the `graph` member function `lastResult(cout)` to output the parent pointers (recall that both algorithms set parent pointers to encode the MST and the SSSP, respectively). You can follow the example of `graph_main2.cpp`. For Prim's algorithm, generate an undirected graph and for Dijkstra's algorithm, try both directed and undirected graphs. This can be done by a Boolean parameter in the `generate` function (the default is for undirected).

Problem 2
Prove using a cut-and-paste argument that a minimum spanning tree, not only minimizes the total weight, but also minimizes the maximum-weight edge.

Problem 3
We have n chess players. We also have m matches that are pre-determined. Each match is between two players. In a match, and in addition to the rules of chess, one player must wear a blue shirt, and the other a red shirt. Players cannot change shirts during the tournament. Describe a linear time algorithm $O(n+m)$ that assigns the shirt colors to the players such that every match will comply to the color rule. If that's not possible, your algorithm should also indicate that. For example, if (A, B), (B, C), and (A, C) are three matches among players A, B, and C, then you cannot assign the shirt colors.

Problem 4 (this is a known classical setting)
You have a lot of foreign currency that you want to convert to US dollars through a series of currency trades, so as to maximize the amount of dollars you end up with.

Assume you have n currencies and rates $r_{i,j}$ of exchange between currency i and currency j. Assume also that going around any directed cycle of trades doesn't give you profit. Describe a polynomial time algorithm to figure out the best sequence of currency exchanges to maximize your US dollar amount.

Problem 5
Given a graph that is a DAG, design an efficient algorithm that finds if there is a vertex u such that every vertex is reachable from u.