CSC172 LAB 20

DIJKSTRA'S ALGORITHM

1 Introduction

The labs in CSC172 will follow a pair programming paradigm. Every student is encouraged (but not strictly required) to have a lab partner. Labs will typically have an even number of components. The two partners in a pair programming environment take turns at the keyboard. This paradigm facilitates code improvement through collaborative efforts, and exercises the programmers cognitive ability to understand and discuss concepts fundamental to computer programming. The use of pair programming is optional in CSC172. It is not a requirement. You can learn more about the pair programming paradigm, its history, methods, practical benefits, philosophical underpinnings, and scientific validation at http://en.wikipedia.org/wiki/Pair_programming.

Every student must hand in their own work, but every student must list the name of their lab partner if any on all labs.

This lab has six parts. You and your partner(s) should switch off typing each part, as explained by your lab TA. As one person types the lab, the other should be watching over the code and offering suggestions. Each part should be in addition to the previous parts, so do not erase any previous work when you switch.

The textbook should present examples of the code necessary to complete this lab. However, collaboration is allowed. You and your lab partner may discuss the lab with other pairs in the lab. It is acceptable to write code on the white board for the benefit of other lab pairs, but you are not allowed to electronically copy and/or transfer files between groups.

2 Weighted Shortest Path

Previously, you will implemented a basic graph ADT using the adjacency matrix data structure. In this lab, you will implement the weighted shortest path algorithm. This algorithm is discussed in detail in section 9.3.2 of your assigned reading (Weiss). A pseudocode description of the algorithm is given in Figure 9.31 of your text, and is repeated below. Use this approach to solve the weighted shortest path problem.

```
// Pseudocode for Dijksra's algorithm (taken from Weiss Fig 9.31)
void dijkstra (Vertex s)
{
    for each Vertex v
    {
        v.dist = INFINITY;
        v.known = false;
    }

    s.dist = 0;

    while (there is an unknown distance vertex)
    {
        Vertex v  = smallest unknown distance vertex;
        v.known = true;
        for each Vertex w adjacent to v
            if(!w.known)
            {
                cvw = cost of edge from v to w;
                if (v.dist + cvw  < w.dist)
                {
                    // update w
                    decrease (w.dist to v.dist + cvw);
                    w.path = v;
                }
            }
        }
    }
}
```

1. Make certain that your implementation of the graph ADT is functional. You will be extending theses classes to add weighted depth first search functionality.
2. Create text files describing the weighted graph in figure 9.20 in your textbook (Weiss) as well as 3 additional graphs of your own design and equal complexity. You will need to extend the file format you used in the last lab to store information about the weight of each edge, and modify the `createFromFile()` factory method to support reading in edge weights.

   At least one of the graphs you include should have an edge with a negative weight. Use the `createFromFile()` factory method to create Graph objects from these files and print them to the console.

3. Implement Dijkstra's algorithm using the pseudocode given at the beginning of this lab. You will have to make a number of design decisions in order to get this algorithm to work with your ADT, including how to store the per-vertex information without using a Vertex class.
4. Test your implementation of the shortest path algorithm on the graphs you made in part 2. Print out examples of shortest paths found using Dijkstra's to the console. Comment about what happens if you try to compute a path with a negative path weight in your README file.
5. Provide a runtime analysis of your code and discuss in your README how your implementation compares to the theoretical running time of the pseudocode presented in the text book.

3 Hand In

Hand in the source code from this lab at the appropriate location on the BlackBoard system at my.rochester.edu. You should hand in a single zip file (compressed archive) containing your source code, README, and OUTPUT files, as described below.

1. A plain text file named README that includes your contact information, your partner's name, a brief explanation of the lab (A one paragraph synopsis. Include information identifying what class and lab number your files represent.), and one sentence explaining the contents of all the other files you hand in.
2. Several Java source code files representing the work accomplished for this lab. All source code files should contain author and partner identification in the comments at the top of the file.
3. A plain text file named OUTPUT that includes author information at the beginning and shows the compile and run steps of your code. The best way to generate this file is to cut and paste from the command line.

4 Grading

Each section (1-5) accounts for 18% of the lab grade (total 90%)

README file counts for the remaining 10%