ECE 223: Computer Systems Engineering **Fall 2013**
Machine Problem 6 **Due: 11:59 pm, Thursday, November 14**

The goal of this machine problem is to implement a module for Dijkstra's algorithm, and to apply the module to solve a variety of shortest path graph problems.

Use a layered design to develop a clean interface to a module that implements Dijkstra's algorithm. You must design and document your interface, clearly defining the required inputs, data structures, and output for the module. You must also design test drivers and implement the user interface as described below. In addition to submitting all source code you must also submit a makefile that compiles all code and creates an executable with the name "lab6". Your code must have no compiler warnings and no memory leaks or other valgrind errors.

Two additional documents must be submitted. One is a **test plan** that describes details of your implementation and demonstrates, with a test script, how you verified that the code works correctly. The verification should include detailed prints from your program to show that you program operates correctly. The second document describes your **performance evaluation**, and the details are described below.

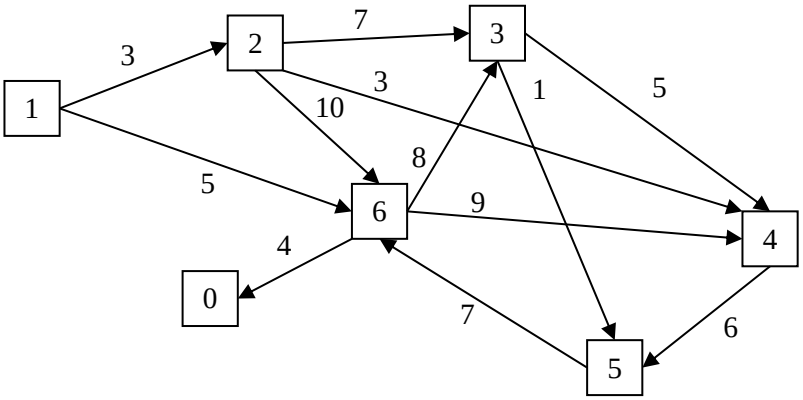
Interface specifications

Command line arguments must be used to set run-time parameters. Use the `getopt` function to implement processing of the command line arguments. See MP4 and MP5 for examples using `getopt`.

-g [1 | 2 | 3]	Graph type
-n *N*	Number of vertices in graph
-a *A*	Approximate number of adjacent vertices $0 < A < N$
-h [1 | 2]	Graph operation
-s *S*	Number of the source vertex $0 \leq S < N$
-d *D*	Number of the destination vertex $0 \leq D < N$
-v	enable verbose output
-r 1234567	seed for the uniform random number generator

Graphs Types (-g [1 | 2 | 3])

1. Use the weakly-connected directed graph shown in the figure below. The –n and –a options are not used.



2. Strongly-connected directed graph with *N* vertices. The vertices are numbered 0 through *N* – 1, where *N* is set by the command line with the –n option. The weight of the edge from vertex *i* to vertex *j* is

$$w_{i,j} = \begin{cases} |i-j| + (i-j+3)^2 + 5j & i \neq j \\ 0 & i = j \end{cases}$$

Notice this graph is not symmetric, and there is an edge between every pair of vertices.

3. Random graphs, with *N* vertices and approximate number of adjacent neighbors *A* (–n *N* –a *A*). The vertices are numbered 0 through *N* – 1.
   - Vertex 0 is at location (0.5, 0)
   - Vertex *N* – 1 is at location (0.5, 1.0)
   - All other vertices have a random (*x*, *y*) location, uniformly located in a square of size 1. That is, for each vertex *v* set *v.x* = `drand48()` and *v.y* = `drand48()`.
   - The distance between vertex i and vertex j is $D(i,j) = \sqrt{(i.x - j.x)^2 + (i.y - j.y)^2}$
   - For a vertex *v*, the average number of adjacent vertices in a large random graph is approximately $A = N\pi C^2$, where C is the radius of a circle centered at *v* and a vertex must be located in the circle to be a neighbor. Since we are specifying *A*, the radius (or normalized communication range), *C*, is

$$C = \sqrt{\frac{A}{N\pi}}$$

   - The weight of the edge from vertex *i* to vertex *j* is

$$w_{i,j} = \begin{cases} 0 & i = j \\ \dfrac{M}{F(i,j)} & if\ D(i,j) \leq C \\ \infty & otherwise \end{cases}$$

   - For a wireless network, we can approximate the (normalized) communication rate between two radios as $F(i,j)\ /\ M$, where M is the minimum data rate when the two radios are at the maximum separation for which communication is acceptable, and $F(i,j)$ is the data rate than can be achieved when the radios are closer to each other than the maximum separation. We can approximate these values as

$$M = \log_{10}\left(1 + \left(\frac{1}{C}\right)^2\right)$$

and

$$F(i,j) = \log_{10}\left(1 + \left(\frac{1}{D(i,j) + \frac{C}{1000}}\right)^2\right)$$

Notice this graph is symmetric ($w_{i,j} = w_{j,i}$), but the graph may not be connected. Generally, for moderately large graphs there is a chance that a path between each pair of vertices cannot be found if the average number of neighbors, *A*, is smaller than approximately 10. With high probability the graph is connected if *A* is larger than approximately 20.

For random graphs you **must** print the following information after the graph is built: The average, maximum, and minimum number of adjacent vertices taken over all *N* vertices.

Graph Operations (-h [1 | 2])

1. Shortest path. Given input parameters -s *S* and -d *D*, find and print the cost of the shortest path from *S* to *D* and the vertices in the path in order from the source to the destination. If there does not exist a path from the source to the destination, print a message instead of the path.

   For example, if the parameters at –g 1 –h 1 –s 1 –d 5, your program should print that the cost of the path is 11 and the path is 1 -- 2 -- 3 -- 5. If the parameters are –g 1 –h 1 –s 5 –d 1, your program should print that a path does not exist between these vertices. For –g 2 –n 100 –h 1 –s 0 –d 99 the cost is 2,768 and there are 9 vertices in the path (there are multiple paths with the same cost, and the path you find depends on how ties are broken).

2. Network diameter. If the graph is connected, find the source-destination pair (*s*, *d*) for which the cost of the shortest path from *s* to *d* has the largest value among all possible values for *s* and *d*. If the graph is not connected, then the output should print the value for the source-destination pair that is connected and has the largest cost, and must print that the graph is not connected. The –s and –d command line parameters are not used. For example, for –g 1 –h 2, the diameter is 21 for the path 4 -- 5 -- 6 -- 3, and the graph is not connected.

Performance Evaluation

1. Node density for random graphs. For the types of random graphs generated with the –g 2 option, there is a relatively narrow range for the value of C (the radius of a circle that determines communication range) such that the graph changes from a low probability of being connected to a high probability. The graph operation –h 2 (network diameter) is one easy way to determine if a graph is connected. Design an experiment to determine the probability that a graph is connected given a value for *A*, the approximate number of neighbors. Because the graphs use random locations for the nodes, for a given value of *A*, you need to generate at least 10 different graphs by using 10 different seed values (you set the seed with –r seed), and calculate the fraction of the graphs that are connected. Collect results for at least 5 different values of *A*. The lowest value of *A* must be a value such that the fraction of graphs that are connected is less than 20%, and the highest value of *A* must be a value such that the fraction of graphs that are connected is greater than 80%. At least three values for *A* must show fractions that are between 20% and 80%. You can select any value you like for the number of nodes, *N*, provided it is greater than 99. For example you might use: -g 3 –h 2 –n 100 –a *A* –r seed, for your values of *A* and the different seeds. Create a plot to show the probability that a random graph is connected versus values for *A*. You should create a shell script, such as we did in MP3, to collect the data.
2. Computational complexity verification. For two graph types (-g 2 –h 1 –n *N* –s 0 –d *N* – 1 and –g 3 –a 15 –h 1 –n *N* –s 0 –d *N* – 1), measure the time to calculate the cost of the shortest path using the `clock()` function. Find the smallest graph size, N_1 so that the time to find the shortest path is just greater than one second for source 0 and destination $N_1 - 1$. Plot the run time versus *N* for values between $N_1/10$ and N_1. Discuss the computational complexity class of Dijkstra's algorithm. Note that graph type 2 is a worst case since each vertex has $N - 1$ neighbors, so the number of edges is $O(N^2)$. Graph type 3 has a fixed number of neighbors for each vertex on average so the number of edges is $O(N)$. Does the difference in number of edges for the two graph types change the computational complexity for your implementation? (Hint: probably not unless you use an advanced data structure, such as a heap representation of a priority queue, for storing and updating the `ShortestDistance` vector).

Notes

1. The system files #include <limits.h> and <float.h> have definitions for maximum and minimum numbers. Be very careful if you use INFINITY. This symbol is defined in <math.h> and if used in a calculation can "overflow at translation time". Be very careful if you use a negative number as infinity. Searches for the largest or smallest value become very confusing. For floating point numbers, use double instead of float. A convenient large number is FLT_MAX (~e38). Using DBL_MAX (~e308) is also possible but can lead to overflow problems that are hard to find.
2. You must choose the approach for representing a graph, using either an adjacency matrix or adjacency list. If you use a matrix, see http://c-faq.com/aryptr/dynmuldimary.html to review how to allocate two-dimensional arrays. For example:

```
    double **array = (double **) malloc(GraphSize * sizeof(double *));
    for(i = 0; i < GraphSize; i++)
        array[i] = (double *) malloc(GraphSize * sizeof(double));

  Gives array[x][y] for 0 <= x < GraphSize and same for y
```

3. It can be dangerous to compare floating point numbers for equality. If you assign a constant to a floating point number and never change it, then you can test if it is still equal to the constant. However, if any computation is performed on the variable, then comparisons for equality often fail (and can be machine dependent). For example,

```
    double x = FLT_MAX;

This works if x is never changed:
    if (x == FLT_MAX)  // true
However,
    double y = sqrt(2.0);
    x = x / y;
    x = x * y;
then tests for equality can fail:
    if (x == FLT_MAX)  // may not be true
```

See the ECE 223 Programming Guide for additional requirements that apply to all programming assignments. All code and documentation files must be turned in by email to assign@assign.ece.clemson.edu. Use as subject header ECE223-1,#6 . **Work must be completed by each individual student, and see the course syllabus for additional policies**.